Embedded Systems

Ultra-low energy domain-specific instruction-set processors

Francky Catthoor · Praveen Raghavan
Andy Lambrechts · Murali Jayapala
Angeliki Kritikakou · Javed Absar

Springer

Ultra-Low Energy Domain-Specific Instruction-Set Processors

Francky Catthoor • Praveen Raghavan
Andy Lambrechts • Murali Jayapala
Angeliki Kritikakou • Javed Absar

Ultra-Low Energy Domain-Specific Instruction-Set Processors

Springer

Prof. Dr. Francky Catthoor
Interuniversity MicroElectronics Center
IMEC
Kapeldreef 75
3001 Leuven
Belgium
catthoor@imec.be

Dr. Praveen Raghavan
Interuniversity MicroElectronics Center
IMEC
Kapeldreef 75
3001 Leuven
Belgium

Dr. Andy Lambrechts
Interuniversity MicroElectronics Center
IMEC
Kapeldreef 75
3001 Leuven
Belgium

Dr. Murali Jayapala
Interuniversity MicroElectronics Center
IMEC
Kapeldreef 75
3001 Leuven
Belgium

Eng. Angeliki Kritikakou
Univ. Patras
VLSI Design Lab
26110 Rio
Greece

Dr. Javed Absar
Samsung India Software
Operations Pvt. Ltd
No. 66/1, Bagmane Tech Park
C.V. Raman Nagar
Bangalore - 560 093
India

ISBN 978-90-481-9527-5 e-ISBN 978-90-481-9528-2
DOI 10.1007/978-90-481-9528-2
Springer Dordrecht Heidelberg London New York

Library of Congress Control Number: 2010932574

Cover design: eStudio Calamar S.L.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Modern mobile embedded systems offer a wide range of features. These devices are capable of offering various applications. They can also communicate to different devices using different wired and wireless standards. For a good battery life for these devices, they have to be extremely energy efficient. All parts of such a device have to be optimized to reach an acceptable energy efficiency. This book focuses on the platform architecture of these devices and how to improve the energy efficiency of the processors, the data and instruction memory organisation.

In order to perform a consistent optimization, we describe a complete, consistent framework for architecture exploration. Such a framework allows the designer to get a complete view of the processor and therefore allowing him/her to see the global effect of the optimization instead of a narrow view of only component. This book discusses the elements of the framework but also the main results and guidelines which we have derived from it. Next, we describe the different platform architecture extensions that enable the ultra-low power domain-specific processor realisation. They include the instruction memory organisation of the processor where also efficient multi-threaded execution of multiple loops is enabled. In addition, we propose a novel way to map indirectly indexed arrays and dynamically accessed data on the scratchpad and data memory organisation. And we also propose very energy-efficient foreground memory architecture as alternative to the traditional power-consuming register-files used now. Finally, we have also extended the processor data-path itself by incorporating an effective fine-grained data-parallel support that is fully software controlled by compiler directives.

This book also provides a recent overview of the state-of-the-art literature in energy-efficient platform architectures, intended for portable

data-dominated applications domains. The material that we present here is based on research at IMEC and its university partners in this area in the period 2001–2009.

Our approach is very heavily application-driven which is illustrated by several realistic demonstrators, partly used as red-thread examples in the book. Our target domain consists of embedded systems and especially portable real-time processing systems which deal with medium to large amounts of data. This contains especially multi-dimensional signal processing (RMSP) algorithms like video and image processing in many different application areas including bio-oriented sensors, graphics, video coding, image recognition, medical image archival, advanced audio and speech coding, multi-media terminals, artificial vision, graphics processing. But it also includes wired and wireless terminals which handle less regular arrays of data. Those are present especially in the digital front-end and baseband signal processing, including the cognitive radio sensing engines. Other target areas include automative signal processing and bio-informatics.

The cost functions which we have incorporated for the data and instruction memory organisation are access rate, memory footprint and energy consumption. For the data-path we focus on area cost and energy consumption. Due to the real-time nature of the targeted applications, the latency or throughput is normally a constraint. So performance is in itself not an optimisation criterion for us, it is mostly used to restrict the feasible exploration space. The potential slack in performance is used to optimize the real costs like energy, area, and (on- or off-chip) memory foot-print.

We therefore expect this book to be of interest in academia, both for the overall description of the exploration methodology and for the detailed descriptions of the main ultra-low energy platform contributions. The priority has been placed on issues that in our experience are also crucial to arrive at industrially relevant results. All projects which have driven this research, have also been application-driven from the start, and the book is intended to reflect this fact. The real-life applications are described, and the impact of their characteristics on the methodologies and platform components is assessed.

We therefore believe that the book will be of interest as well to senior architecture design engineers and their managers in industry, who wish either to anticipate the evolution of commercially available design concepts over the next few years, or to make use of the concepts in their own research and development.

The material in this book is partly based on work in the context of several European and national research projects. And it has also been indirectly sponsored by the industrial IMEC program partners in the wireless and design technology activities. Also the Flemish IWT has sponsored some of the work in regular IWT (especially SWANS) and SBO (especially Flexware) projects.

It has been a pleasure for us to work in this research domain and to co-operate with our project partners and our colleagues in the low power platform architecture and embedded compiler community. Much of this work has been performed in tight co-operation with many university groups, mainly across Europe. In addition to learning many new things about system synthesis/compilation and related issues, we have also developed close connections with excellent people. Moreover, the pan-European aspect of the projects has allowed us to come in closer contact with research groups with a different background and "research culture", which has led to very enriching cross-fertilization. This is especially reflected in the many common publications. We want to especially acknowledge the valuable contributions and the excellent co-operation with: the ACCA group at K.U.Leuven (now part of ELECTA at the El.Eng.Dep. of the K.U.Leuven, Belgium), U.Autonoma Barcelona (Spain), U.Completense Madrid (Spain), Osaka Univ. (Japan), Patras Univ. (Greece).

We would like to use this opportunity to thank the many people who have helped us in realizing these results and who have provided contributions in the direct focus of this book, both in IMEC and at other locations. That includes everyone who helped us during the PhD. and M.S. thesis research.

In particular, we wish to mention: David Atienza, Francisco Barat, Bruno Bougard, Erik Brockmeyer, Henk Corporaal, Stefan Cosemans, Geert Deconinck, Hugo De Man, Veerle Derudder, Bjorn De Sutter, Bert Geelen, Jos Huisken, Jos Hulzink, Yuki Kobayashi, Rudy Lauwereins, Anthony Leroy, Toon Leroy, Min Li, Stylianos Mamagkakis, Pol Marchal, Dragana Miljkovic, Satyakiran Munaga, David Novo, Martin Palkovic, Antonis Papanicolaou, Antoni Portero, Christophe Poucet, Adelina Shickova, Guillermo Talavera, Ittetsu Taniguichi, Christian Tenllado, Karel Van Oudheusden (Edgar Daylight), Arnout Vandecappelle, Liesbet Van der Perre, Tom Van der Aa, Diederik Verkest, for all their technical feedback during the research that is incorporated in this book. Indirectly also many others have contributed in some way though and the list would be too long to explicitly mention them all here. But we do acknowledge their support.

Also thanks to all the masters students who helped in the research: Chaitanya, Estela, Georgia, Ioannis, Jean-Baka, Nandu.

We finally hope that the reader will find the book useful and enjoyable, and that the results presented will contribute to the continued progress of the field.

The book authors: Francky Catthoor, Praveen Raghavan, Andy Lambrechts,
Murali Jayapala, Angeliki Kritikakou, Javed Absar.
Autumn 2009

Contents

Glossary and Acronyms

3GPP	3rd Generation Partnership Project
ADRES	Architecture for Dynamically Reconfigurable Embedded System
ADOPT	ADdress OPTimisation
AGU	Address Generation Unit
ALU	Arithmetic and Logic Unit (A functional unit that can perform arithmetic and logic operations)
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction set Processor
AVC	Advanced Video Codec
BB	Basic Block
COFFEE	COpiler Framework For Energy-aware Exploration
CGRA	Coarse Grained Reconfigurable Array
CORDIC	COordinate Rotation DIgital Computer
CPU	Central Processing Unit
CSD	Canonic Signed Digit
DEMUX	De Multiplexer
DL	Data memory Level
DLP	Data Level Parallelism
DM	Data Memory
DMA	Direct Memory Access
DPG	Data path generator
DSP	Digital Signal Processor
DTSE	Data Transfer and Storage Exploration
EPIC	Explicitly Parallel Instruction Computing
FEC	Forward Error Correction
FF	Flip Flop
FU	Functional Unit
GCC	GNU Compiler Collection

GPP	General Purpose Processor (e.g. an Intel Pentium processor)
GSA	Generic Shift Add
GSAS	Generic Shift Add Shift — Generic Shift Add Shuffler
HRF	Hierarchical Register File
ICMO	Instruction/Configuration Memory Organization
IL1	Instruction Level 1 Cache
ILP	Instruction Level Parallelism
IM	Instruction memory
IP	Intellectual Property
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
KB	Kilo Bytes
LB	Loop Buffer
LBL (1)	Linearly Bounded Lattice (polyhedra context)
LBL (2)	Local Bit Line (memory context)
LC	Loop Controller
LD	Elementary Load operation
LSB	Least Significant Bit
LUT	Look Up Table
MIMO	Multiple Input Multiple Output
MPEG	Moving Picture Experts Group
MSB	Most Significant Bit
MUL	Multiplier
MUX	Multiplexer
NOP	No Operation
Op	Operation
PE	Processing Element
PC	Program Counter
P & R	Place and Route
RF	Register File
PR	PLR Pipeline Register
RISC	Reduced Instruction Set Computer
RF	Register File
RLB	Real Lower Bound
SDR	Software Defined Radio
SDRAM	Synchronous Dynamic Random Access Memory
SIMD	Single Instruction Multiple Data
SoftSIMD	Software SIMD
SoC	System on a Chip
SMT	Simultaneous Multi Threading
SRAM	Static Random Access Memory
SRF	Scalar Register File
SA	Shift Add
SAS	Shift Add Shift
SSA (1)	Static Single Assignment (polyhedral context)

SSA (2)	Shift Shift Add (ASIP context)
TTA	Transport Triggered Architecture
TLP	Task Level Parallelism
UUB	Upper Upper Bound
VHDL	VHSIC Very High Speed Integrated Circuits Hardware Description Language
VLIW	Very Long Instruction Word (Also abbreviation for VLIW processor)
VWR	Very Wide Register
WLAN	Wireless Local Area Network
WiMAX	Worldwide interoperability for Microwave Access
ZOL	Zero-overhead Looping

CHAPTER 1

Introduction

Abstract

This book focuses on domain-specific instruction-set processor architecture exploration for embedded systems. Next to a general framework to explore such platforms, we will also propose different platform architecture extensions that enable ultra-low power domain-specific processor realisation.

This introduction describes the general context of this book (Section 1.1), in which some open problems are identified. Section 1.2 discusses the focus of the book. The key elements addressed here are listed in Section 1.3. Finally, Section 1.4 explains the structure of the rest of the text.

1.1 Context

Modern consumers carry many electronic devices, like a mobile phone, digital camera, GPS, PDA and an MP3 player. The functionality of each of these devices has gone through an important evolution over recent years, with a steep increase in both the number of features as in the quality of the services that they provide. Access to multimedia content and wireless communication have become ubiquitous. At the same time, Swiss-knife-like units, combining a range of functionalities into one compact package, have become widespread.

F. Catthoor et al., *Ultra-Low Energy Domain-Specific Instruction-Set Processors*,
DOI 10.1007/978-90-481-9528-2_1,

However, providing the required compute power to support (an uncompromised combination of) all this functionality is highly non-trivial. Continuing the trend of increasing quality (e.g. higher resolutions) and wider feature sets, in increasingly compact units leads to complex designs. It requires the development of embedded compute platforms that can combine the flexibility to accommodate the different and often dynamically variable performance requirements with the extremely high energy efficiency needed for a battery-powered device.

In addition to these established markets, emerging applications in domains like wireless sensor nodes, smart medical implants or precision agriculture have even more constrained requirements and present interesting new opportunities for embedded processing. The work that is presented in this book mainly targets this type of mobile, battery operated embedded systems. The individual elements can be reused in other low-energy domains also though.

Visionaries have emphasized and re-emphasized the human urge for ever more ambient intelligence to make improve various aspects of human life ranging from communication, health care, safety and others. This quest of Ambient Intelligence (AmI) has necessitated higher computation and higher communication requirements in electronic devices. This “dream” or vision requires that there is secure computing and communication embedded in everything and everybody and it is context/ambient aware and sensitive to the person/user.

For realizing such an ambitious dream a wide range of electronic devices is needed. This space of electronic devices is often split in three large classes: stationary devices, nomadic devices and transducer-based devices. These three classes of devices have different power-performance requirements. The three different classes with their requirements are shown in Figure 1.1. At one extreme of these devices range are “sensor nodes” which perform little computation and have a very low power budget of few tens of microwatts. At the other extreme are “stationary devices” connected to the power supply which perform very high computation and have a power budget of tens to hundreds of watts. In the middle exist the “nomadic devices”, which are mobile handheld devices which are powered by batteries. These nomadic devices have a power budget of few tens to few hundreds of milliwatts and have a modestly high computation and communication requirement.

To reach this dream it is necessary to accommodate the high computational requirement under the power constraints of each of these different classes of devices. This problem is further compounded by the fact that the communication (and consequently computational) data rate has been increasing dramatically and predictably [Che04]. The various number of different types of applications and communication standards that need to be supported on

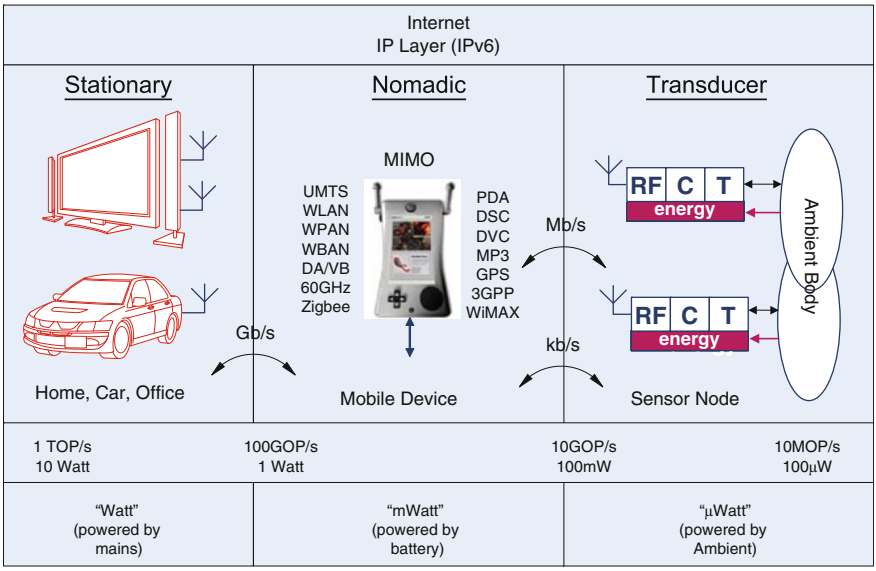


Figure 1.1: Different device class in the context of a dream of Ambient Intelligence (as described in [DeM05])

these devices also require that the device is flexible. However poor scaling of battery technology is not helping towards reaching this goal. This requires the designer of such devices to manage multiple objectives: supported applications, performance, flexibility, area and energy efficiency.

In order to achieve this we have to create large improvements on the energy-efficient realisation of all the platform components. In this book we will focus on a single (data-parallel) processor core so the main components involve the processor itself, its data memory hierarchy and the instruction memory organisation. We will not look at the (inter-processor) communication organisation. For the background data memory hierarchy we will mainly reuse state-of-the-art work in terms of software controlled scratchpad memories both in terms of the architecture aspects and the mapping/compilation issues [Cat98b, Ver07]. So this book is mainly focused on the processor-related components including the local instruction memory organisation.

1.1.1 Processor design: a game of many trade-offs

Traditionally, the combination of energy efficiency and high performance requirements has been a synonym for Application Specific Integrated Circuits or ASICs, which are completely optimized for a single application (Figure 1.2).

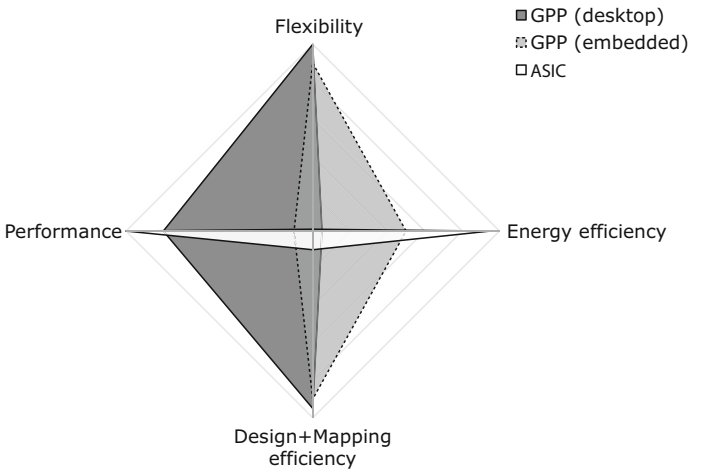


Figure 1.2: Characterization of desktop General Purpose Processor (GPP), embedded GPP and ASIC, according to multiple design criteria

However, the increasing cost of designing specialized chips in modern Deep Sub-Micron (DSM) technologies, together with the need to fit more functionality into a single system, has forced designers to build more flexible solutions. If evolving product generations and a wide range of similar products can use the same chip, non-recurring engineering costs can be amortized over larger markets. This aspect clearly favors programmable or configurable components. in the context of embedded processors. The use of a processor, especially off the shelf components, has an added benefit that it can help to reduce the time to market, which is extremely important in a consumer electronics context.

Designing processors that meet the demanding requirements of future mobile devices requires the optimization of the embedded system in general and of the embedded processors in particular, as they should strike the correct balance between flexibility, energy efficiency and performance. A complete range of solutions has been proposed, giving different weights to each criterion in this trade-off space and, in addition, requiring different amounts of design time or mapping (implementing an application on a processor) effort. Classes of processors can be distinguished from each other, based on the amount of parallelism they exploit, the way parallelism in applications is detected, the amount of customization to a certain application or domain, etc. Each class will hit a different balance in the trade-off space that is discussed here.

This book mainly focuses on the domain of nomadic embedded processor platforms. In that context, Figure 1.2 presents a comparison of two well-know

classes of processors to an ASIC solution, with respect to four important criteria, namely flexibility, performance, energy efficiency and the design/mapping efficiency. A General Purpose Processor or GPP is a very flexible processor (can run any application), as is used in desktop or laptop computers. The desktop GPP is an off-the-shelf product and applications can be easily mapped using push-button compilers. Therefore, the design/mapping efficiency is a strong point. However, the rather high performance and emphasis on flexibility comes at a clear penalty. A desktop GPP typically consumes tens of watts and hence does not score well on the energy efficiency axis. Due to the energy-constrained nature of battery operated mobile embedded systems, simple devices often make use of very simple small embedded GPPs, which are mostly sequential RISC processors. They can run many different applications at much better energy efficiencies, but have a limited peak performance that is often not sufficient for our target application domain. The design/mapping efficiency is still high, as very mature compilers exist for these systems.

As embedded systems typically do not suffer from code compatibility issues and do not need to support legacy code (for which only binaries are available) as is common for general purpose computing, the compiler can be modified from one device to another and from one version to the next. Specific compilation techniques and manual optimizations can be used to match the application and the processor design in the best way possible. This freedom also enables the use of architecture extensions and processors, which can be designed for specific application domains. In this way, different classes of embedded processors trade-off design/mapping efficiency to improve both the performance and energy efficiency. In the context of battery operated embedded systems, classes that push the trade-off towards energy efficiency, without loosing all flexibility, are especially worth looking at. One interesting class of processors that follows this philosophy is the Very Long Instruction Word processor or VLIW [Fis05]. VLIWs perform multiple operations in parallel in order to deliver the required compute power for e.g. multimedia applications and make use of a compile-time detection of this parallelism. Generic VLIW processors still have a flexible instruction set, but by adding special instructions and features, both the performance and energy efficiency can be heavily improved for specific application domains. This results in Application Specific Instruction-set Processors (see ASIP in Figure 1.3). Adding more resources, as in the case of Coarse Grained Reconfigurable Processors or CGRAs, improves the performance and potentially also the flexibility further, but this can reduce the energy efficiency. Some applications require extremely energy efficient solutions (Ultra Low Power, as in the ULP-ASIP of Figure 1.3), but more customization (e.g. in the data and instruction memory organizations) will be required to achieve this. As a result, both the flexibility and design/mapping efficiency move to less optimal points.

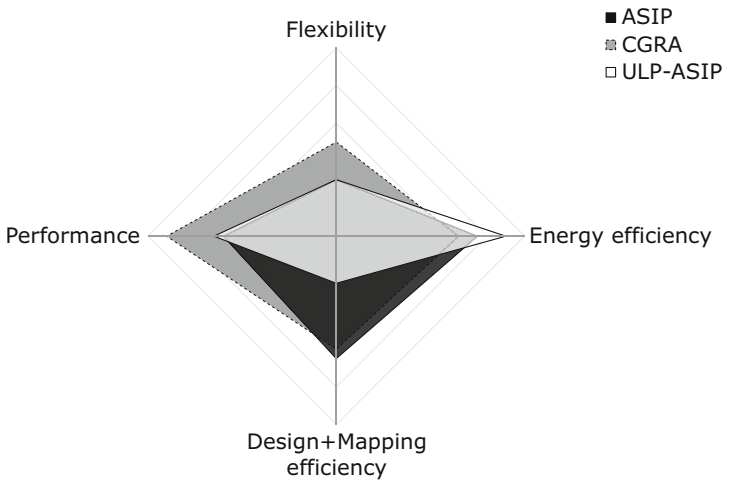


Figure 1.3: Characterization of CGRA, ASIP and ULP-ASIP according to multiple design criteria

1.1.2 High level trade-off target

The overall embedded processor design space is extremely large. The classes of processors that have been described above, are often not clearly defined in literature and many hybrids exist. In general, a designer will try to $minimize\ the\ energy\ consumption$ (as far as needed) for a $given\ performance$, with a $sufficient\ flexibility$. However, achieving this goal is already complex when looking at the processor in isolation, but, in reality, the processor is a single component in a more complex system. The complete embedded platform typically consists of multiple processors, each optimized for a set of tasks, together with dedicated accelerators, memories and interconnection networks (e.g. Ti's OMAP and DaVinci platforms [TI09c]). In order to design such complex system successfully, critical decisions during the design of each individual component should take into account effect on the other parts, with a clear goal to move to a global Pareto optimum in the complete multi-dimensional exploration space. This book is mainly focused on the processor and its local instruction memory organisation (see above). Still, to avoid a too narrow viewpoint, in Chapter 3 we will also put our work in a more global perspective by analyzing the performance and energy contributions of the different platform components for different architecture styles.

1.2 Focus of this book

The target scope of this book is the class of embedded systems, that are usually also imposing stringent (soft or hard) timing and performance constraints. The main subclass for which we illustrate our concepts are "nomadic devices" that require both flexibility and increased energy efficiency. However principles used in this book are extensible for other embedded systems in domains like video and image processing in many different application areas including bio-oriented sensors, graphics, video coding, image recognition, medical image archival, advanced audio and speech coding, multi-media terminals, artificial vision, graphics processing.

Typically a nomadic device would consist of various different parts: user interface (screen), wireless communication interface (antenna), and digital and analog chips. Given the high communication data rate and the computation requirement, the design of digital components within the energy budget of the battery has become a challenging task.

To perform the digital computation in these nomadic devices, often unflexible ASIC solutions are used and at times power-hungry programmable processor solutions are used. Both these solutions give a trade-off between energy efficiency/performance and flexibility. However the raising number of applications and communication standards have necessitated the need for flexible solution to be both cost effective as well as reduce time-to-market.

Various design options exist for such flexible processors based solution. Figure 1.4 shows a normalized graph of energy efficiency of the processor to their peak performance. Note that most processors in this graph belong to the class of nomadic hand-held processors and each have different architectural parameters. Each of these architectures gives a different trade-off between energy efficiency, performance, programmability and flexibility.

To meet the every increasing computation and communication requirement, very high energy efficiency is needed. And to reach this high energy efficiency, it is important to keep both the technology aspect as well as the application into mind during the design of the processor core. As we scale to smaller and smaller technologies, scaling issues like domination of interconnect energy play an important role. Therefore the design of the processor architecture must take these effects into account. As the title of the AmI dream suggests: the device has to be "ambient" aware. This implies that the application adapts to the ambient. Adaptation to the environment makes the application dynamic. The processor design must also take these application characteristics into account. Given that current processor architectures do not reach the required high energy efficiency, future architectures and compilation techniques need to take into account the application and technology aspects. Chapter 3 performs an in-depth analysis of various technology as

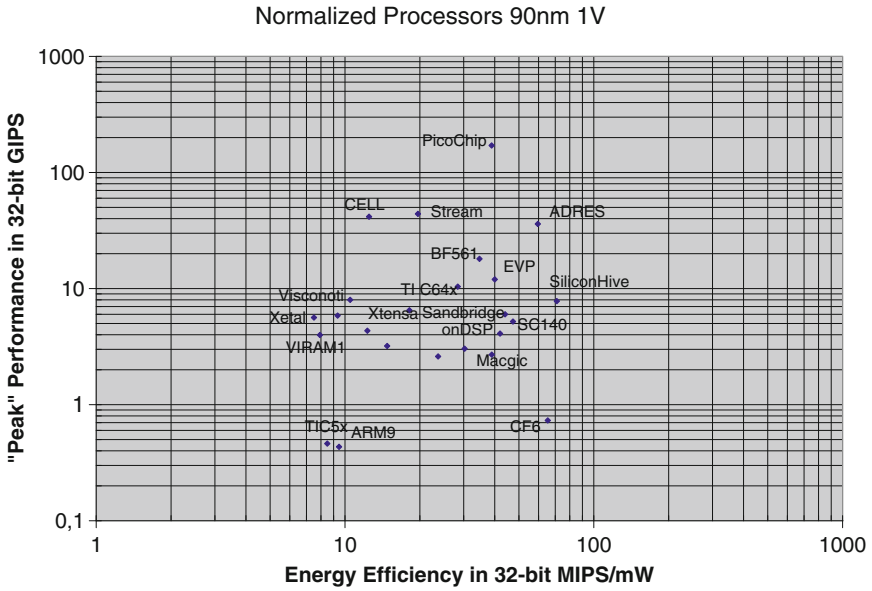


Figure 1.4: Comparison of efficiency of different state of the processors

well as application characteristics which must be taken into account during the design of an energy-efficient processor core.

In the complex, global design of battery-operated embedded systems, the focus of this book is on the $energy-aware\ architecture\ exploration$ of $domain-specific$ processor datapaths and the co-optimization of the datapath architecture with its mapping techniques or $compiler$ at $early\ stages$ of the design.

- Architecture Exploration: In order to balance application, architecture and mapping, architecture exploration is used as a method to estimate the quality of the design with respect to various quality criteria.
- Energy-aware: Even at early stages of the design, where the impact on the final energy efficiency is still large, energy-estimation is essential and should give a clear overview of the relative contributions of different components in the platform.
- Domain-specific: A clear motivation exists to push for the development of energy-efficient and performant processor based solutions, optimized for a range of applications, in contrast to application specific solutions or accelerators.
- Compiler: The ability to efficiently map applications to a processor is key to the success of the designed solution. Exploiting more application information during the mapping and looking at mapping techniques

and datapath design together, gives rise to new opportunities and trade-offs. This compiler is however not the main focus of this book. See [Rag09b, Lam09] for more information on this important topic.

- Early design stages: A clear benefit exists in getting feed-back on many quality criteria, including energy consumption, early on in the design. At the higher abstraction levels (at Register Transfer level, before gate level), different design styles can be compared relatively fast. The impact of this decision potentially has a large impact on the quality of the final implementation.

Processor architecture exploration is often considered in isolation from the rest of the platform. By explicitly taking into account the relations with other components of the platform, the risk of optimizing one component at the expense of negatively affecting the performance and energy-consumption of others is tackled. In the processor architecture exploration itself, a trade-off exists between the accuracy of the energy consumption estimation and the abstraction level at which it is conducted. Often energy estimation is postponed until late in the design, when a detailed and accurate estimation is possible, but many processor design decisions are already fixed. Intra-processor interconnect energy consumption is mostly neglected during the early stages of the processor design. However, for some more recent classes of processors, the amount of intra-processor interconnect is much larger than for others, resulting in a clear underestimation of the energy consumption at the higher abstraction levels. Especially in more scaled technologies, the wire dominance (in energy consumption) is further increasing, so this effect will become more pronounced. In general, the processor energy estimation accuracy vs. speed trade-off is currently not optimal for performing systematic architecture exploration.

From a compiler point of view, extra optimizations can be added by exploiting application knowledge and by adapting the mapping to the dynamism of the application. Modern applications often exhibit highly variable performance requirements in different phases of their execution. At the application level, this can often be linked to different accuracy requirements and this information is currently not fully exploited in the final implementation. Clear opportunities exist to couple application accuracy requirements and dynamism to the word-width of the data and the exploited parallelism. Additionally, during the architecture exploration, the instruction set definition and mapping techniques can be co-designed in order to reduce the total execution cost, as not all operations have equal energy consumption, latency or area.

The related work in the context of embedded systems design mostly focuses on one component, e.g. the processor [Abb07] or the data memory organization [Ram05]. With respect to the relations with other components, default solutions are used, instead of globally combining multiple state-of-the-art aspects across components. This leads to skewed interpretations

of improvements and decisions that are often not globally improving the design. That applies for domain-specific processors in general. Specifically for Coarse Grained Reconfigurable Processors (CGRAs), only a few proposed architectures are explicitly template based [SilH, Ebe96] and support exploration and customization to the target application domain. Some limited parameters have been explored, e.g. [Ban04] investigates the impact of different network topologies for mesh-based CGRAs, but the template is too restricted and the work is for performance only. No existing work enables early and systematic energy-aware and interconnect-aware exploration over the template. Note that CGRAs could be seen as quite similar to the family of instruction-set programmable architectures when the semantics (not the syntax) of the constituting components is analyzed.

Also for the ASIP research community, a similar observation can be made: individual contributions are available on the components, but no global holistic energy optimisation is enabled in this way. Still, the interest in ASIP style processor platforms has become very high in the last 2 decades, especially in the last few years. Early advocates in the 1990s of that approach were Masaharu Imai [Sat91], Peter Marwedel [Mar03], Gert Goossens [Goo95], and Pierre Paulin [Pau00, Poz07], together with their teams. In a broader context of domain-specific processors, architecture exploration and the development of architecture extensions for specific application domains is closely linked to the ability of compilers to efficiently use these extensions. Therefore, many opportunities exist at the architecture–compiler interface. Some limited attempts have been made to link application requirements and efficient implementations through the used of variable word-widths [Eve01, Ber06]. However, also no systematic overview has been presented with respect to exploiting word-width information throughout the mapping process, trading off execution cost with application requirements. The same observation specifically also applies for the exploitation of strength reduction techniques for multiplications in the context of a complete and broad search space. Current compilers only focus on a part of the available exploration options, for a specific context only [Har91, Par01].

Chapter 2 provides a more thorough review of the related work in these domains.

1.3 Overview of the main differentiating elements

The differentiating elements presented in this book are closely linked to the main problem and subproblems that have been identified in the previous section and are all linked to the architecture exploration of the embedded

processor datapath, extended with its compilation techniques. The research results that are presented in this book have been obtained in the context of a team of Ph.D. students that have been active both at IMEC and several university groups with which we collaborate. The result is hence a team outcome.

1. A comprehensive energy consumption breakdown is provided for an embedded platform, running a representative application, including a detailed analysis on the relative importance of the different platform components and on the way they influence each other. The study includes a comparison of different embedded processor styles, individually optimized for the same application. This breakdown indicates that, although the datapath does only account for a small percentage of the total energy cost, the processor design is an important part of the system, as its design heavily influences the cost of the data and instruction memories [Lam04, Lam05, Lam09, Rag09a, Rag09b].

2. A key element in this book is the detailed analysis of the space of embedded processor architectures and its relation to process technology and application domains [Lam09, Rag09b].

3. This analysis has lead to the proposal of a parameterized template of a domain-specific instruction-set processor platform, called FEENECS, which offers an energy-performance figure of merit of over 900 MOPS/mW in the TSMC 40 nm CMOS technology (using non-optimized standard cell synthesis). This is less than a factor 2 higher than an optimized ASIC for the same application instance, as illustrated in [Kri09].

   Importantly, the basic architectural components of the FEENECS template have a quite wide scope including all domains that are loop-dominated, that exhibit sufficient opportunity for data level parallelism, that comprise signals with multiple word-lengths (after quantisation exploration) and that require a relatively limited number of variable multiplications. Prime examples of this target domain can be found in the areas of wireless base-band signal processing, multimedia signal processing or different types of sensor signal processing [Lam09, Rag09b, Kri09, Rag06a].

4. A consistent architecture exploration framework is proposed which can model a large set of architectural features for low power design. It forms one of the key elements in this book. This framework has also been extensively used in the rest of the book as well as to perform architecture exploration over different architectural parameters to illustrate the different trends and trade-offs that are present [Lam09, Rag08a, Rag08b, Rag09b].

5. A distributed instruction memory organisation is introduced based on distributed loop buffers. Also the interaction with a clustered foreground data memory organisation is discussed [Rag06a, Jay02a, Jay02b, Jay05a, Vda03, Vda04a, Vda04b, Vda05].

6. An efficient low power multi-threading architecture extension is proposed for embedded processors. This architecture extension allows the possibility of running multiple threads with communication and synchronization with nearly no overhead. This is enabled by providing an alternative distributed instruction memory organization. The proposed approach also increases performance and reduces energy consumption [Rag06a, Rag09b, Sca06].

7. Handling irregular indexes and dynamic accesses is a key issue for the background data memory organisation. Ways to effectively deal with this on a compiler-driven scratchpad architecture are proposed [Abs08, Abs05, Abs06, Abs07].

8. Another core element is a novel asymmetrical foreground memory (register file) architecture for embedded processors. The proposed architecture extension is shown to be energy efficient as well as improve performance by exploiting the spatial locality of data in the application [Rag06a, Rag07a, Rag09b].

9. A systematic methodology is proposed for exploiting (heterogeneous and non-power of 2) word-widths during mapping applications to embedded processors, namely during assignment, scheduling, ISA selection and parallelization, including word-width aware energy estimation at the Instruction Set Simulator level and a quantification of the expected gains for the different proposed techniques. Based on the estimated gains, parallelization is identified as the most promising technique, as it affects the energy consumption in all parts of the processor and the platform, while the other techniques are most useful in systems in which the datapath energy cost is a bottleneck [Lam07, Lam09].

10. A technique for word-width aware parallelization is described, operating on different word-widths in parallel, without using special hardware (Software SIMD). The Software SIMD technique provides significant speed-ups and a reduction in the energy consumption for applications that have heterogeneous word-widths or in which the word-width is significantly different from the hardware supported word-widths [Lam07, Kri09, Lam08a, Lam09, Rag06b, Rag07b].

11. A cost-driven method for constant multiplication strength reduction is proposed, including a systematic and complete description of the conversion space. The unique benefits of this method are the co-optimization with the processor instruction set and a trade-off of

accuracy at the application level, which can lead to a reduction in the energy consumption, a performance improvement or both. The presented technique can be used as an enabling step for Software SIMD [Lam08b, Lam09].

12. All the key domain-specific instruction-set processor/platform extensions are illustrated on a realistic demonstrator, namely an online bioimaging monitoring application. There also the large gains in energy-efficiency for a given (high) performance are quantified and substantiated. Due to our architectural innovations, we obtain about 1,000 MOPS/mW for a 40 nm TSMC CMOS technology using a conventional standard cell library combined with a low-power SRAM macro. This is a factor 10 higher than state-of-the-art processors that span a similar target domain [Kri09].

1.4 Structure of this book

The flow of the book as well as the dependencies of the different chapters are shown in Figure 1.5. The organization of the rest of the book is as follows:

Chapter 2 presents a structured overview of the global related work. It briefly discusses the important architectural components of battery operated embedded systems and the related mapping techniques. It discusses architecture exploration and evaluation methods, together with the most relevant design criteria or cost metrics. The specific related work for each key element and comparisons with our proposals will be presented at the end of each of the following chapters.

Chapter 3 discusses a case study that presents an energy consumption breakdown for a complete embedded platform, consisting of the processor, data memory hierarchy, instruction memory organization and communication network. The main objectives of this case study are to get a better insight into platform energy estimation in general, to highlight the bottlenecks for this representative platform and to track the effects of local changes on the other parts and the final platform quality. The experiment is conducted using a representative driver application and provides a context to the optimizations that are presented in the rest of the book.

Chapter 4 presents the framework that has been used for architecture exploration in the rest of the book. It also presents exploration of various architectures and illustrates the various trade-offs between different cost criterion like area, energy and performance between different architectures.

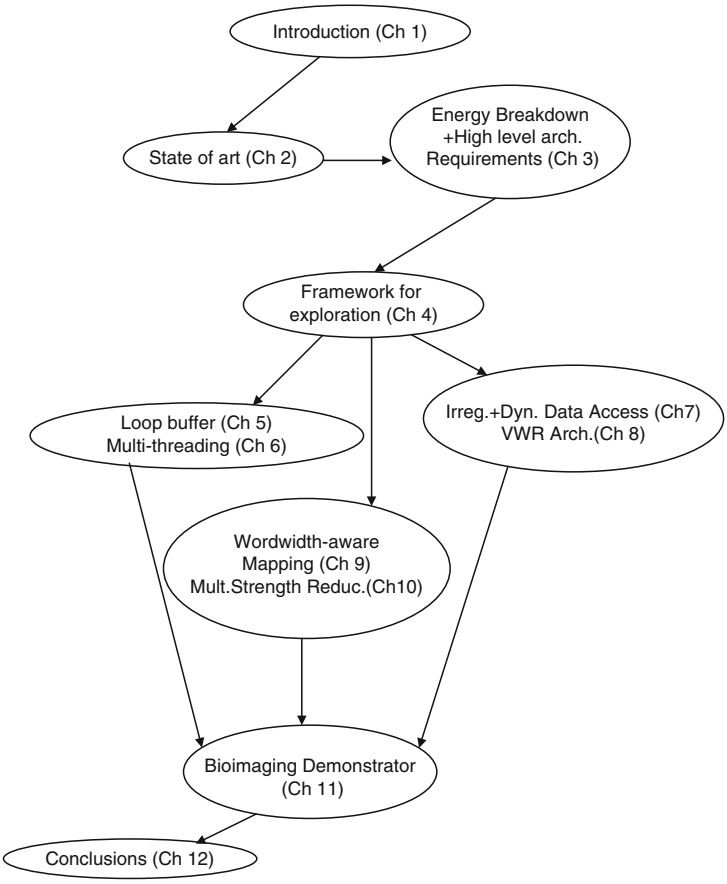


Figure 1.5: Dependence flow of this book

Chapter 5 presents our distributed loop buffer organisation and the integration with the clustered foreground data memory organisation.

Chapter 6 introduces the technique of executing multiple independent threads on a single-threaded processor in an efficient way for both performance as well as energy. The proposed solution ensures that the instruction memory is efficiently organization to enable such multi-threading.

Chapter 7 describes how irregular indexes and dynamic accesses can be handled on the background data memory organisation. Ways to effectively deal with this on a compiler-driven scratchpad architecture are proposed.

Chapter 8 introduces the novel Very Wide Register (VWR) based architecture as an alternative to traditional multi-port register file based design. It also evaluates the VWR architecture over different benchmarks. A case study of the VWR is also illustrated in Chapter 11 for an online bioimaging monitoring application.

Chapter 9 focuses on exploiting word-width information during application mapping in order to reduce the energy consumption or improve the performance of processor-based embedded systems. It presents the use of word-width aware energy models to improve ISS-based energy estimation sensitivity to word-width variation. It then systematically describes how to exploit this information during various steps of the mapping process, namely during assignment, scheduling, ISA selection and parallelization. For each part, the concept of the optimization is detailed and the expected gains are evaluated. This chapter also presents a more detailed description of how to implement word-width aware parallelization, also called Software SIMD.

Chapter 10 details the proposed method for the strength reduction of multipliers. It first motivates that constant multiplications form a relevant sub-set of the multiplications in the targeted application domains and then presents a systematic overview of the complete conversion space. A context-aware cost-driven search over this space is proposed. Experimental results are presented that illustrate the resulting conversion in a set of relevant contexts.

Chapter 11 shows how are proposed architecture innovations can be combined in a very effective ultra-low power template that has been instantiated for realistic bioimaging application.

Chapter 12 draws the final conclusions and presents directions for future research.

CHAPTER 2

Global State-of-the-Art Overview

Abstract

The work presented in this book targets nomadic battery operated embedded systems. In this context, a large amount of related work exists. The goal of this chapter is to present a structured overview of the relevant related work in the design of embedded systems, which forms the broad context. The presented ordering will cover both the architectural as well as the related mapping aspects. An overview will be presented of the state of the art for the different components that form an embedded system. Specific related work and comparisons the individual contributions of this book will be presented in the respective chapters.

The rest of this chapter is structured as follows: Section 2.1 present an overview of the architectural components, namely the processor core, Data Memory Hierarchy (DMH), Instruction/Configuration Memory Organization (ICMO) and the inter-core communication architecture. Section 2.2 introduces the related work on the architecture exploration over this space which forms a key aspect of the embedded systems design, together with the evaluation methods and relevant criteria or cost metrics. Finally Section 2.3 concludes this chapter.

F. Catthoor et al., *Ultra-Low Energy Domain-Specific Instruction-Set Processors*,
DOI 10.1007/978-90-481-9528-2_2,

2.1 Architectural components and mapping

Figure 2.1 shows the main components of an embedded system. At the heart of the system, the *processor core* performs the computations of the application. It operates on the application data, which are stored in the *Data Memory Hierarchy*. The DMH can consist of multiple individual memories of different sizes. The type of operations that need to be executed by the processor and their required order is stored in the *Instruction/Configuration Memory Organization*. As with the DMH, the ICMO can consist of multiple memories. Modern embedded systems often contain multiple processor cores and use an inter-core *communication architecture* to connect all components and enable the data transfer between different processors and memories.

The rest of this section will discuss the relevant related work for each of the components of Figure 2.1, including the techniques that are used to map applications efficiently to these architectural components.

2.1.1 Processor core

The processor core consists of the hardware that executes the operations (the datapath), the foreground memory from which the operands are loaded and to which the results are stored back and the local interconnection between these components. In addition to these components, the processor core also consists of other components like processor pipelining and the issue type. In this book, the foreground memory is defined as the memory to and from which reads and writes happen in the same cycle as the processing, e.g. register files, pipeline registers. Related work on these individual parts can be structured according to these components as shown in Figure 2.2. Based on the design decisions made for each component, processor styles can be defined. The state-of-the-art processors representative for those styles, will be categorized and described and the end of this subsection.

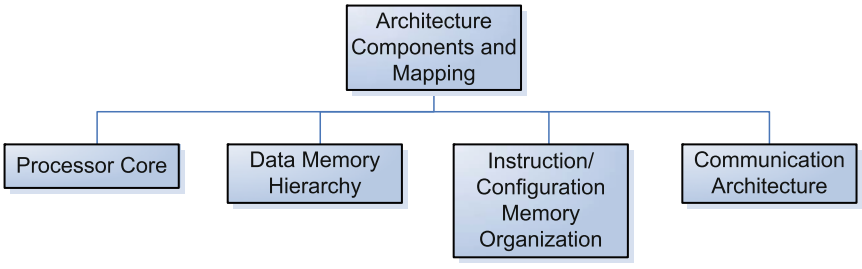


Figure 2.1: Processor architecture space

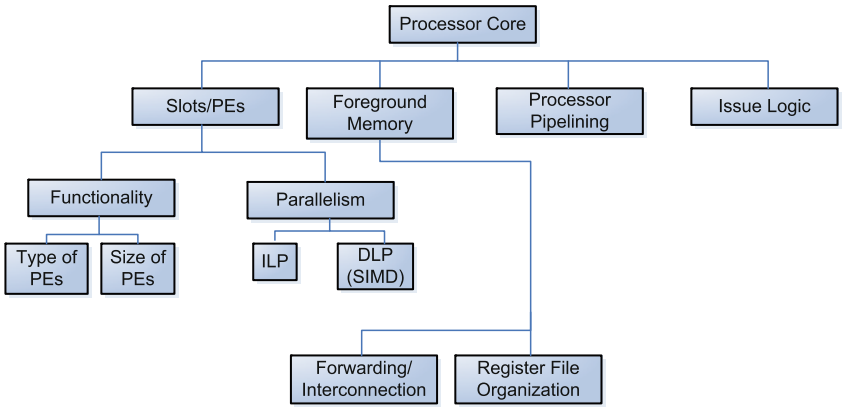


Figure 2.2: Processor core architecture space

2.1.1.1 The FUs, slots and PEs of the datapath

Depending on the community, the terms FU, slot and PE can be used to represent the same or different architectural components. Therefore a clear definition is presented below, that is followed throughout the rest of this book.

Functional Unit (FU) The hardware component that executes an operation of a specific type, e.g. and adder, multiplier or shifter. A single-issue processor can still contain multiple FUs, but then only one FU can start execution in a certain cycle.

Slot A group of FUs that are used mutually exclusively. Multiple (issue) slots can execute in parallel. The terms *slot* is commonly used in Very Large Instruction Word (VLIW) processor literature.

Processing Element (PE) Similar to a slot, but can also contain a local data or instruction storage. The term PE is commonly used in reconfigurable hardware literature.

Operation The functionality that is executed by an FU, e.g. ADD, MUL, SHIFT.

Instruction The binary representation that determines what operation will be executed by a single slot or PE. In VLIW literature, the VLIW-instruction is a concatenation of the instructions for all parallel slots.

In conclusion, the slots or PEs of a processor typically consists of different types of Functional Units and each FU supports a different functionality. The

instructions decide what *operation* is executed on a certain slot and on the corresponding FU within that slot. Typically the combination of FUs into slots or PEs is called the processor *datapath*.

Processors can differ in the type and mix of FUs and in the number of slots. In order to be able to efficiently execute their target application domains the designer has to match these parameters to the requirements of the application. The set of operations that is supported by the FUs of a processor is called the *instruction set*. Various techniques exist that identify customized instruction set extensions and implement them [VPr94, Che05, Lee03, Yu04] . These techniques identify appropriate sub-graphs from the control-data flow graph that can be executed by a single customized FU and in a way that efficiently accelerates the application.

The number of slots of the processor determines the amount of instructions that can be issued in parallel (Instruction Level Parallelism or ILP). Various processors offer different levels of instruction level parallelism [TI00, Phi, VdW05]. A range of techniques exist to compile to architectures with multiple issue slots. For example modulo scheduling or hyper-blocking [Ron07, Rau94, Mah92] increase the number of instructions that can be scheduled together.

Irrespective of the number of parallel slots (the ILP), each of the different FUs of the processor can potentially support the execution of the same operation on multiple data words in parallel. This is called SIMD (Single Instruction Multiple Data) and is a form of Data Level Parallelism or DLP. Various processors offer different possible SIMD possibilities [Fre, Int, TI00, VdW05]. The individual data elements on which the operation is performed in parallel are called *sub-words* and together they form a *word*. In order to exploit the DLP that is present in the application, often the data layout (the order in which the data is stored in the memory) has to be modified and the data sub-words that will be operated in parallel have to be (re-)packed together into words. This process, together with the re-structuring of the application code to perform the data parallel execution is called vectorization or SIMDization. Various SIMDization and transformations techniques have been proposed in literature [Bou98, Xue07, OdB03, Lar05].

2.1.1.2 Foreground memory (or register files)

The second component of the processor core is the foreground memory, from which the data is loaded and to which the data is stored back after the execution. The *foreground memory organization* consists of the Register File (RF) and the connectivity between the RF and the FUs or between different FUs (forwarding/interconnection) (see Figure 2.2).

The biggest challenge when designing a register file is to balance the need to deliver data in parallel to all slots with the requirement to keep the number of ports low for a feasible implementation and higher energy efficiency [Rix00a, Lap02]. Clustering (splitting into parts) of the register file is a typical technique to allow delivering data in parallel to many slots in a scalable way without increasing the number of ports per register file. However the communication between the different clusters can have a negative effect on performance. An extensive study of various inter-cluster communication architecture is presented in [Gan07].

Other types of foreground memory organizations have also been proposed, ranging from hierarchical register files [Zal00a] to FIFO based register files [Tys01, Fer97]. Vector register files [Kap03, Asa98, Koz03] target data paths that provide data level parallelism (SIMD) and form another important class of foreground memory.

For each of these architectures a range of register allocation techniques exist [Zha02, Smi04, Cha82]. Some techniques like [Das06] target specific streaming models of register file as well.

In addition to the register file and the datapath, most state-of-the-art processors have special forwarding paths between the functional units. This allows more flexibility as data can be sent from one FU to another without passing through the register file [Gan07, Sch07]. As the availability of forwarding paths has an impact on the register allocation, the compiler which maps to these architectures must also support such forwarding paths.

2.1.1.3 Processor pipelining

Processor pipelining is a design method that inserts an extra register between different phases of the processor execution. Typically different stages are used for Fetch, Decode, Execute and Writeback [Hen96]. As a result the critical path is shortened, which allows increasing the clock frequency. In order to reach the required clock frequency for current designs, state-of-the-art processors have a pipelining depth ranging from 3 to 12. However, increasing the frequency arbitrarily will lead to the additional stages and an increase in the dynamic energy/power consumption due to the extra registers. In extremely scaled technologies, timing differences due to process variation lead to even more design problems for very high clock frequency processors. Therefore the processor design needs to carefully balance the exploited parallelism and target frequency (linked to the pipelining depth) in order to reach the required performance.

2.1.1.4 Issue logic

Different processor types can be categorized based on the order in which they issue and complete their instructions. Some processors support the out-of-order issuing of instructions. Out-of-order execution requires the hardware to keep track of what hardware is currently in use, which instructions have been executed, which operands have been produced etc. Alternatively, the responsibility of keeping the hardware busy can be moved to the compiler, in which case, in-order issuing can be used and no expensive hardware is required. With respect to the completion of the instructions, two types exist. With in-order completion, the results need to be written back in the same order as the instructions have been issued. With out-of-order completion, this restriction is removed, which leads to more flexibility, but a higher complexity.

Traditionally techniques like speculative execution, out-of-order issue and completion are extensively used in high performance super-scalar processors e.g. Intel's series of x86 processors, Power PC. In battery operated embedded systems, due to the energy efficiency requirement, a preference is given to simple hardware and the issue (and completion) order of instructions is fixed by the compiler at compile time. Using techniques like software pipelining, the execution can follow a different order than is given by the DFG, thereby creating freedom to optimize the performance. This allows much lower overhead at run-time and therefore is more energy efficient.

2.1.1.5 Overview of state-of-the-art processor classes

The embedded processor space contains a wide range of options. On one side, very low power processors target battery operated or even self-sustaining systems, but provide only a limited performance. On the other side, very parallel types can provide an extremely high performance in case a larger battery or a connection to the power grid is acceptable. The state-of-the-art embedded processors can be split up into different processor types, each offering a different trade-off point in the performance vs. energy efficiency vs. mapping effort space. At the lowest performance and lowest flexibility side of the processor spectrum, ultra-low power micro-controllers, like TI's MSP430 [TI09a] can be used to do very basic types of processing. They are good candidates when the total energy budget is extremely constrained and when the workload and performance requirements are correspondingly low. They are namely optimized mostly for control tasks.

Slightly more flexibility can be provided by small sequential RISC processors, e.g. [ARM09a]. They are typically small in-order single slot machines that have a shallow pipeline and that can exploit only a limited amount of parallelism.

Another class of ultra low power processors are targeting sensor nodes (as shown in Figure 1.1). These processors have a power budget in the range of tens to hundreds of microwatt. Examples are [Eka04, Kar04, Cal05, Naz05]. But also in that case, the maximal performance is heavily limited. Especially at U.C. Berkeley [Rab0-] major efforts have been invested to motivate the need to improve the energy efficiency to reach the levels of these scavenging limits, and to contribute in these improvements for sensor node networks.

By providing multiple slots, VLIW processors can increase the performance, while minimizing the required hardware overhead (compared to superscalar processors). Low power embedded VLIW processors can typically execute between 2 [TI09b] and 8 [TI06] operations in parallel. Most of them combine this however with a limited form of data parallelisation on several of the slots (e.g. the C6x series of TI and the TriMedia of NXP [VdW05]). So every instruction then executes several operations in parallel (e.g. 2 tot 4). This can potentially increase the maximal performance significantly. Many VLIWs provide a rather general instruction set and therefore are still quite flexible. But on the other hand, most of them are quite optimized for executing digital signal processing tasks in e.g. wireless or media processing.

Another class of VLIW style processors is organized as wide or hierarchical VLIW processors, which provides more flexibility than pure vector processors, as different operations can be executed in parallel [SilH, Mon05]. They form very heterogeneous VLIWs.

Some VLIW processors however support only quite specific operations that improve the performance for a selected target application or application domain. Various processor extensions (SIMD, loop buffering, clustering, etc.) can be used to improve energy efficiency, performance or both. When the processors are more customized in this way, they become an Application Specific Instruction set Processor or VLIW-ASIP. In this case a distinction can be made between ASIPs and accelerators, e.g. [Lu99]. An accelerator is customized to accelerate only part of the application, while the rest is executed by a so-called host processor. An ASIP often combines both into one processor.

Worth mentioning in the wireless domain are the following ASIPs. NXP's embedded vector processor (EVP) is a software-programmable platform for basebands with a relatively wide data-path [Kum08]. Infineon's MuSIc (Multiple SIMD Cores), is a DSP-centered and accelerator-assisted architecture and aims at battery-powered mass-market handheld terminals ([Ram07]) The University of Michigan has developed a very low-power processor meant especially for relatively low performance sensor applications – called Phoenix [Woh08]. It consumes power as low as 30 pW. Sandbridge [Glo04,

Sandbridge] has launched a multi-core, multi-threaded, dynamically reprogrammable processor that supports SIMD vector processing. At the Univ. of Dresden, the Tomahawk platform and its predecessors have been introduced [Hos04]. And also at the University of Aachen, ASIP related work has been ongoing for several years [Schl07].

In addition, several commercial activities on such ASIP IP cores have been launched. Some of these are also template-based. In particular, Tensilica's Xtensa and Lx processors belong to this family [Xtensa]. And Silicon Hive has proposed several VLIW-ASIP cores oriented to multi-media and wireless processing too [Phi]. Also CSEM's Macgic DSP template belongs to this category [Ram04].

Heterogeneous cores, like [Bar05b], combine multiple styles into a single processor, e.g. a RISC core with a more parallel datapath part, to accelerate the execution of specific application parts only when needed.

To meet the real-time constraints of some highly regular application domains (limited control flow) with high throughput requirements, extremely parallel processors have been developed. These processors are called vector processors. Some make extreme use of SIMD like [Abb07].

Others are explicitly targeting graphics applications, can handle many parallel threads with low overhead and are called GPUs or Graphics Processing Units for e.g. [Nvi09, ATI09]. These GPUs also offer very high SIMD and accelerators for various graphics operations. They also contain extra hardware to support the dynamism of fast thread creation and management used in, e.g. objects in 3D gaming.

Finally some embedded processors extensively make use of data communication between their PEs in order to map larger parts of the data flow graph onto the processor. These processors are often organized as a 2D array, and are usually called Coarse Grained Reconfigurable Architectures or CGRAs, e.g. [Mei03a, PAC03]. CGRAs offer a wide range of design parameters, like the type and topology of their interconnect, the number of slots, etc.

Figure 1.4 provides a comparison of several of the main processor classes and a few representative instances in the energy-performance trade-off range.

In order to achieve an efficient mapping of software onto this wide range of processors, various mapping and compilation techniques have been developed. This ranges from the efficient extraction of data parallelism, thread level parallelism, exploiting ILP and various other techniques. As architecture and compiler are closely linked, most of the above mentioned references include a description on relevant compilation strategies for the respective processors.

2.1.2 Data memory hierarchy

Next to the processor core, the data memory hierarchy is a second component that has a high impact on the performance and power consumption of the platform. In general purpose computing the movement of data between different parts of the data memory hierarchy is often handled by hardware support. This corresponds to the use of so-called hardware controlled *caches*. In most embedded applications the data access pattern is more regular and can be analyzed. Therefore the movement of data can be more efficiently handled by the compiler or programmer and scratchpads are used (data memory without hardware support to move data around). Hybrids also exist, in which part of the cache can be used as a scratchpad, e.g. in [VdW05]. A scratchpad based solution is power and performance efficient [Kan04a, Ban02], but it requires a DMA or Direct Memory Access, which is a separate datapath that needs be programmed to handle the required transfer. [Tal08, Mat03, Het02] present various architectural possibilities and their trade-offs for DMAs and Address Generation Units (AGUs) [Tal08].

Besides the aspect of the type of memory that is used (hardware controlled cache or scratch pad), it is also possible to exploit the reuse of data over time and the access order pattern (locality). Therefore a right choice of the memory sizes, multiple levels and their inter-connectivity (memory hierarchy) is needed for low power and high performance. The locality can only be efficiently exploited if the application is transformed to make the most efficient use and transfer of the data possible. Various transformation techniques that expose the data locality (both spatial and temporal) have been proposed in the literature [Abs08, Bro00a, Mar03, Pan98].

2.1.3 Instruction/configuration memory organization

The instruction memory contains the bits which describe the functionality of the different slots and other processor components. In the CGRA/FPGA community this is also called the configuration memory. They are conceptually similar and are treated together in this book. Instructions/configurations are distributed and stored in the Instruction/Configuration Memory Hierarchy or ICMO.

A good ICMO aims at distributing the instructions over the architecture at the appropriate time and at a low cost. The ICMO can be organized similar to the data memory hierarchy, namely as multiple hierarchical layers. However, instructions differ from data in some important aspects (e.g. they are only read, not written) and therefore some extra optimizations can be used. With respect to the locality, instructions offer extra opportunities for (mostly temporal) reuse as most embedded applications consist of computations that are

organized in the form of loops. Various techniques like filter caches [Kin00], trace caches [Rot96], victim caches [Jou90, Jou94] have been proposed to optimize the behavior of caches for loop code.

Further optimizations, like loop buffers [Kav05, Uh99] go one step further, by adding a special scratchpad (software controlled) to store the instructions of loops only. Loop buffers can also be clustered [Vda04b, Jay05b] and their size can be customized to reduce the energy/power consumption.

Independent of the fact if hardware controlled caches or scratchpads are used, the instructions stored in memory can be compressed. This allows a smaller memory foot print and is especially important for very parallel architectures, e.g. VLIW processors. These techniques also reduce the traffic between the instruction memory and the processor cores, but require an additional decompression before the actual execution step. Various techniques of instruction compression and NOP compression exist in literature [Deb00, Tan02, Gor02b, Adi00]. A detailed survey of instruction compression is given in [Bes03].

A more detailed overview of the design space of instruction/configuration memory organization is presented in Chapter 6.

2.1.4 Inter-core communication architecture

In order to use all described components in an embedded system, they have to be connected. This can be done using different architectures, ranging from custom point to point connections, over a standard bus (e.g. ARM's AMBA), to a network-on-chip. An overview of state-of-the-art communication architectures can be found in [Kog06, DeM06, Ler08] for network-on-chip solutions or [Pap06, Hey09] for bus based work. The communication architecture can be chosen based the traffic requirements, number of components, design time, etc. A description of the trade-off space for communication architectures is outside the scope of this book and is not discussed further.

2.2 Platform architecture exploration

The range of embedded processors is large and both architectural as well as mapping extensions can be used to modify a processor style. Therefore an exploration phase should be used to find the best match between architecture and application. Conceptually, the architecture exploration can be spit into three different parts, as is shown in Figure 2.3. Firstly, a *search strategy* has to be chosen to go over the different candidates. Secondly, a number of

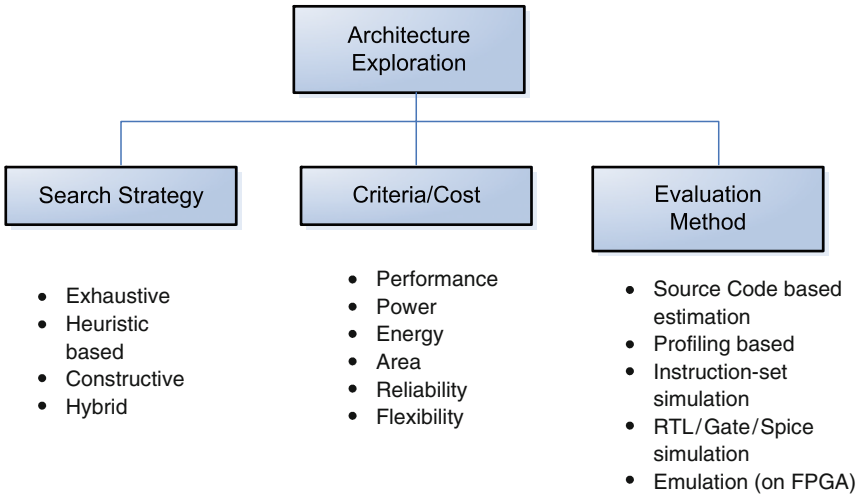


Figure 2.3: Architecture exploration tools/models

criteria or cost estimates, on which each candidate architecture has to be evaluated on. These are often cost models which may be analytical or empirical. Finally, each candidate solution has to be evaluated using a certain *evaluation method*, with respect to the design criteria. These three parts are discussed in more detail below.

2.2.1 Exploration strategy

In order to design an embedded processor that meets all requirements with respect to performance, energy efficiency, area, etc. architecture exploration is used to compare a range of architecture candidates and gradually improve the design. An exploration strategy or technique is needed to steer this process, as the exploration space is extremely large. In most practical cases, the exploration is restricted to a part of the space, usually called the supported architecture template [Gon02, Mei03a, Tar08]. This template can be the result of the experience of the design engineer, often including a combination of application knowledge and intuition, or the result of restrictions in the used mapping tools (e.g. the compiler). The exploration within the template can be further steered by a set of heuristics to speed up the exploration further, as in most cases an exhaustive search would still be infeasible. Such heuristic based exploration is performed in commercial tools, like Tensilica's Xtensa [Gon02] and in academic work, like [Jac00, Lap02]. Alternatively, constructive techniques can be used to generate an architecture based on the analysis of a set of applications, usually starting from a minimal (or extremely

rich) base architecture and demand driven adding (or removing) resources. This approach is popular in high level synthesis and in cases where only a fixed set of applications have to be run. An example of such constructive techniques can be found in Synfora's PICO [Syn08, Phi04].

2.2.2 Criteria/cost metric

The target of the architecture exploration is to find an architecture that meets the requirements. These requirements can be defined with respect to a number of criteria or cost metrics, of which performance, energy efficiency, area, design effort, flexibility are the most used examples. Requirements can be a combination of relative (trade-offs) or absolute (constraints) criteria, e.g. minimize the energy consumption within the available design time (trade-off) for a given minimal performance target (constraint). The effect of a modification at the architecture level, be it a change in processor or memory style or an extension to a component of a processor, propagates to the various design criteria in a non-trivial way. This is because all criteria are linked and strong trade-offs exist, either in the design of a single component or elsewhere in the system.

2.2.2.1 Performance

As most embedded systems interact with the physical world, their real-time performance is often essential. Therefore performance is one of the most important requirements. *Simulation* is a commonly used technique to estimate the performance during embedded processor design, e.g. [Aus02]. The processor can be modeled at different abstraction levels and the corresponding simulation can be instruction accurate, cycle accurate or bit accurate. Cycle accurate simulations are commonly used to estimate the performance of the processor. However, to validate the real-time performance (in seconds), a target clock frequency is needed. To obtain an accurate estimate for this, the complete design needs to be synthesized/placed and routed. The more accurate the simulation needs to be, the slower the process of estimating the performance. In most cases, for relative comparisons a crude estimate suffices however.

2.2.2.2 Energy consumption

In the context of battery-operated embedded systems, trade-offs with respect to the energy efficiency of the architecture can be made visible throughout the design if the energy consumption can be estimated. Traditionally, energy

estimation is done rather late in the design flow, e.g. at gate level. However, this means that many decisions have been taken and a design change at this stage of the design process would require a large effort. Therefore it is difficult to perform a broad exploration at this level or to make significant modifications based on the feedback from the gate level energy estimation. It is however possible to create energy models at various levels of abstraction, which can then be used during exploration at higher abstraction levels. In this way both benefits, namely speed and accuracy can be combined to some extent, which suffices for relative comparisons during early exploration.

Models calibrated after measurement For widely used processors like an ARM9 [ARM09b], TI's TMS320C6x [TI00] series, Intel's processors measurements can be performed to create an accurate energy model. This energy model is often created by a test sequence of instructions or program and measuring the current. Various related work, like [Sin01, Ben02, Jul03, Cha00, Isc03, Tiw96] have created an energy/power model after measurement. These models are often at the granularity of energy consumption per instruction for the complete processor. This complicates the use of the model to make architectural modifications, as they don't produce a breakdown for different processor components.

Analytical models It is also possible to create models that are analytic. Related work like Wattch [Bro00b], CACTI [Shi01] and others like [Mam04, Ye00] create an energy model for individual components. They make assumptions on the micro-architecture for each of the components and on the design style (e.g. standard cell vs. custom design). Based on the technology choice and the micro-architecture assumption, the analytical model gives the energy consumption per activation of the component. Most work in analytical modeling is targeted towards memories as their structure is quite regular and can be captured in a model with reasonable accuracy. However, some work on analytical datapath modeling exists, e.g. [Ask99, Hel02]. The energy models from analytical modeling are often energy consumption per activation of a component. They can also be sensitive to the actual bit switching.

Models at/after RTL Models can also be created at Register Transfer Level (RTL), either for components or for the complete processor. The RTL can be synthesized and the energy consumption can be estimated based on statistical activity. Instead of statistical activity real application data can be used, but this can be very time consuming, especially for a complete processor. The estimation can be made more accurate if done at gate level, and even more accurate after place-and-route and parasitic extraction. A detailed study of the error incurred and the time consumption between the different stages is

done by [Bal04]. Based on the granularity of the modeling, the models will represent the energy consumption per activation of a component or the complete processor respectively. These models can also be extended to take into account the switching activity of the input.

Peak power consumption In the design of embedded systems for mobile devices, energy consumption is an important metric, as it is directly linked to what can be drawn from the battery. However, the peak power consumption has a large impact indirectly, as is influences the worst case heat dissipation and thereby the packaging cost. Estimating the peak power consumption requires, in addition to e.g. the simulation set-up, the availability of worst case test patterns [Hsi97].

2.2.2.3 Area

In the competitive market of consumer electronics, the cost of the design is extremely important and for some designs the cost is directly related to the area of the chip. A first synthesis of individual components gives a first area estimate at the component level. Further placement and routing of components can improve the accuracy of this estimate. However, both methods ignore the overhead/optimization potential of global place and route. Early in the design, a crude estimation suffices to be able to do relative comparisons. For memories this can be often be estimated by analytical models like Wattch [Bro00b] or by using commercial memory generators like Artisan [ARM] or Virage [Vir].

2.2.2.4 Design effort

Aside from the impact different processor styles have on the performance and energy efficiency, there usually is also an effect on the design effort that is to be expected when choosing for a certain style. For example a simple RISC processor, using a push-button compilation approach, will take a much less design effort than the implementation of an ASIP, which requires the manual modification of the application code to make use of the application specific processor extensions. It can be expected that the energy efficiency and performance of the ASIP will be superior to that of the RISC. But when comparing an ASIP solution with a CGRA implementation, the trade-off becomes less obvious. The design effort is part of the overall design trade-off and therefore should be taken into account, but is rather hard to quantify accurately. Therefore in most cases qualitative comparisons are often used and the weight of this criterion with respect to the other cost metrics is up to the designer.

2.2.2.5 Flexibility

A final important aspect of the embedded platform design trade-off is the flexibility of the designed solution. This aspect covers the ability to map other applications to the platform after the design. One extreme case is an Application Specific Integrated Circuit or ASIC, which is completely fixed to a single task, but performs this task very efficiently. On the other side of the spectrum, a general purpose processor can run any application, but at a severe energy and performance cost. The embedded processor space contains multiple in-between solutions, that provide varying levels of flexibility, at various performance and energy efficiency points. However flexibility is a qualitative criteria as often it is a trade-off with efficiency of mapping.

2.2.3 Evaluation method

A key element of a successful exploration is the ability to evaluate different options with respect to the criteria and cost metrics. Depending on the cost metric, different evaluation methods can be used and the evaluation can be (or needs to be) performed at different abstraction levels or with different accuracy. At each of the different levels the cost models can be used to evaluate the quality of the architecture.

Often the initial algorithm development happens in a high level language like Matlab, at which stage the functionality is evaluated. Based on the computational complexity of the application a this level, a first rough estimate can be made of the required peak performance of the processor. It is then refined down to C,[1] which is functionally equivalent to the Matlab reference and now is ready for mapping on an architecture.

The initial architecture exploration is often performed on Instruction Set Simulators. Various simulators exist, like Trimaran [Tri08], Simplescalar [Aus02] etc. Instruction Set Simulators are often accompanied with an associated (retargetable in some cases) compiler front end which compiles the code on the architecture. Commercial instruction set simulators, like Target Tools [Tar08], Processor Designer [CoW08a], Synfora's PICO [Syn08], ASIP Meister [EDA05], Tensilica [Gon02] are available at this level.

Architectures can also be modeled at transaction level. The architecture in this case is modeled in a transaction accurate model often in SystemC or similar languages. Frameworks like Liberty [Lib02], Unisim [Uni05] can be used for this purpose. In practice these simulators model on an abstraction level below ISS level and therefore are more accurate, but slower

[1] The C-language is used most for the embedded systems domain, which is the target of this book, but other languages can be used.

than instruction set simulators. Some commercial frameworks like CoWare's Virtual Platform [CoW08b] also exist to support TLM (Transaction Level Modeling) of such processor architectures together at the platform.

On further refinement, architectures can also be evaluated/explored at Register Transfer Level (RTL). This is very time consuming and often only minor modifications are done during exploration at RTL level at the complete processor level. This can then be refined to gate level and simulated as well. Both RTL and gate level simulation for multi-million gate designs are prohibitively slow and therefore are not usable.

However, instead of simulating the gate level model, it can also be mapped on an FPGA. This is called *emulation* and it is commonly used practice to verify the final design, before producing the chip. Emulation of an architecture is faster than RTL or gate level simulation and therefore some amount of exploration can also be done at this level. Various academic FPGA emulation projects like [UCB07, Ati07] and commercial emulation platforms like Mentor Graphics' VStation [Men07] exist.

2.3 Conclusion and key messages of this chapter

This chapter has presented a structured overview of the related work in the design of battery operated embedded systems, with an emphasis on the design, exploration and evaluation of embedded processors. Specific related work that is directly related to the specific contributions of this book will be discussed at the end of each respective technical chapter.

The design space for embedded processors is extremely large. In addition, the complete embedded platform consists of multiple components (each with a range of options) that can not be studied in total isolation, as modifications to one component can influence the other components. Architecture exploration can be used to find a good match between the application requirements and the architecture design. To be able to perform an efficient exploration and compare different processor styles, this exploration should be performed early on in the design. Therefore a trade-off has to be made between the implementation effort that is required to model an architecture, the evaluation speed and the accuracy. In the next chapters this broad exploration will be performed based on a realistic case study resulting in a set of high-level requirements and a proposal for an ultra-low energy platform template. In addition, a framework will be proposed to effective support this broad exploration.

CHAPTER 3

Energy Consumption Breakdown and Requirements for an Embedded Platform

Abstract

Current embedded systems are built of many interacting components. While optimizing the system, it is important to track the impact of the different parts and their interaction on the global optimality metrics. In this chapter, a representative case study is presented that estimates and compares the most important parts of an embedded platform: namely the processor, data memory hierarchy, instruction memory organization and communication network. The experiment uses a realistic driver application (a MPEG2 video encoder/decoder chain) to estimate the relative importance of the different parts on the final performance and energy consumption of the system. The main objectives of this case study are to get a better insight into platform energy estimation in general, to highlight the bottlenecks for this representative platform and to track the effects of local changes on the other parts in order not to move to a globally worse point. It thereby provides a context to the optimizations that are presented in the rest of this book. Also high-level requirements are derived for the entire platform.

F. Catthoor et al., *Ultra-Low Energy Domain-Specific Instruction-Set Processors*,
DOI 10.1007/978-90-481-9528-2_3,

3.1 Platform view: a processor is part of a system

Future SoC platforms will have to satisfy many critical requirements: they will have to be energy efficient, to be able to handle diverse applications and to provide sufficient processing capacity, all at the same time. The combination of all these requirements poses serious challenges on the current platform styles and related mapping methodologies.

To meet these requirements, most SoCs contain several types of processor cores and data memory units, resulting in very heterogeneous platforms. Current designs are often still organized around a shared bus, a solution that is not scalable on the long term as it leads to a bottleneck in the communication when the number of processors and memories grows. To overcome this problem, other solutions, like Sectioned Buses [Pap06, Hey09] and Network-on-Chip (NoC) solutions [DeM06, Ler08] have emerged, both in academia and in industry. In this chapter, such a state-of-the-art NoC-based SoC is the subject of the presented case study.

As has been shown in Chapter 2, most related work focuses on one aspect of the platform, e.g. the data memory hierarchy, instruction memory organization, communication network or the processor, while a global effort containing all these components is essential to estimate their relative importance and the impact on each other. Additionally, the related work on full-platform estimation focuses on homogeneous platform tiles and regular array/mesh structures, while in reality more heterogeneous platforms are far more likely. The presented experiment aims to expose the real bottlenecks and highlight present trade-offs related to this context. ISS-based energy estimation is used to estimate the energy consumed in the different parts, as has been motivated in Section 2.3. As already indicated in Chapter 1, the multi-processor aspect will not be the real focus of this book. We will look at multi-threading aspects on a single processor in Chapter 6 but not beyond that. The main focus lies on the global interaction between all the key elements of one processor and its data and instruction memory organisation.

Due to the wide scope of this case study (a complete platform) and the large number of components involved, it is not feasible to model all parts to the highest possible detail. Therefore, less important aspects have been ignored or parts that are complex to estimate at a higher abstraction level have been simplified. However, these assumptions are made $explicit$ and are motivated quantitatively during the analysis of the results. This discussion is instructive for understanding the relationship between the different parts of the platform and the conclusions that are drawn about the relative contribution of the parts to the total energy consumption.

In summary, the goal of this case study is to get insight into the relations between and the relative contributions of the platform components and less emphasis should be put on the absolute numbers.

The rest of this chapter is organized as follows. A global experiment based on a MPEG2 video encoder/decoder benchmark allows us to summarize the conclusions for the full system, as presented in Section 3.2. Here the data and instruction memory hierarchy and the processor data-path are viewed in the context of a heterogeneous multi-processor platform including the impact of the inter-processor communication network. Based on this it is clear that the data and instruction memory organisation is dominant initially. Then a more focused exploration of different processor styles is presented in Section 3.3 where it is concluded that the clustered VLIW and CGRA (coarse-grain reconfigurable array) styles are very attractive ones, especially when domain-specific optimisations like custom instruction/FUs and SIMD parallelism exploitation are incorporated. Section 3.4 derives the high-level requirements for an well-balanced low-power platform, and Section 3.5 presents the context of this work and the current trends (including deep-submicron effects) and bottlenecks in the processor architecture exploration space. Section 3.6 presents various architectural proposals for the main platform components, namely data and instruction memory organisation and processor datapaths, in order to reach a similar overall energy efficiency as that of an ASIC. Section 3.7 combines these different architectural components to present the FEENECS architecture template that we propose in this book to achieve ultra low energy execution of embedded loop-dominated applications. Section 3.8 discusses the related work and provides a comparison. Finally, Section 3.9 concludes this chapter and summarizes the key messages of this chapter.

3.2 A video platform case study

The key result in this chapter is that a complete power breakdown of a multimedia platform is presented that, unlike most related work, considers an *optimized mapping* of the application for all platform components. The power consumption of the processor, communication architecture, data memory hierarchy and instruction memory organization are estimated for a representative video processing chain. The relative contribution of different components is studied and relations and the impact of various optimizations are identified.

3.2.1 Video encoder/decoder description and context

The platform considered in this experiment is viewed as a set of tiles interconnected by a communication architecture. A tile can be either a data memory or a compute node, which consists of a processor (e.g. a DSP), including its local data memory.

This section describes the different components of the platform: the driver application, the compute nodes, the heterogeneous platform tiles and the tile sizing and placement. The communication architecture is presented and the mapping of the application on the platform is discussed.

3.2.1.1 Driver application

The power breakdown for a NoC platform is measured for a representative application, consisting of a video chain as is typically used in digital camcorders (see Figure 3.1). This video chain consists of a camera interface (CAM), an MPEG2 encoder and decoder (ENC and DEC), an intermediate buffer (BUF) and a display interface (DISP). Alternatively, the encoded data can be stored to external memory (External). The camera generates 25 4-CIF frames (704×576) per second. This stream is transferred to the MPEG2 encoder. A recent history of the encoded video (a few seconds) is placed in the intermediate buffer to allow the user to quickly playback. When the user wants to replay a recent event, the video is read from this on-chip buffer and sent to the display. This is less energy-consuming than reading from an off-chip device. It is assumed that the entire video is also stored in a dense but slow off-chip storage device for later use. The respective memory interface and additional connections to the off-chip memory are not considered here, but they would not have a significant effect on any of the conclusions.

The bandwidth requirements have been computed for the complete video chain and have been annotated on Figure 3.1. The bandwidth corresponding to the communication between the encoder/decoder and their respective

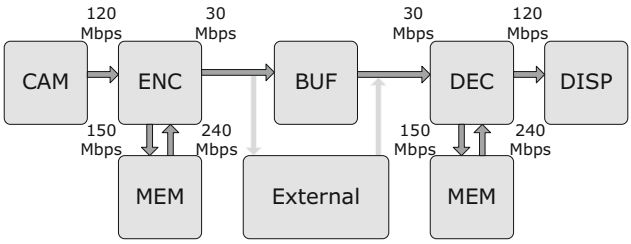


Figure 3.1: Video chain annotated with required bandwidths

data memory tile has been computed with the assumption that extensive Data Transfer and Storage Exploration (DTSE [Moo97]) optimizations have been performed to reduce the number of data memory accesses to a minimum.

3.2.1.2 Embedded platform description

Heterogeneous compute nodes Embedded multimedia applications are very demanding in terms of performance and flexibility and very constrained in terms of energy. They will therefore require both heavily specialized processors and more flexible reconfigurable processors, here in more general terms called **compute nodes**. The correct choice of compute nodes allows these platforms to deliver the correct performance at minimal power consumption, while maintaining some flexibility.

A platform like the one considered in this case study could include a number of compute nodes: a VLIW DSP (TI-C6x-like) for intensive signal processing, a VLIW ASIP (specialized cores) for specific video or audio processing, a Coarse Grained Reconfigurable Architecture (CGRA) for flexibility and hardware acceleration and an ARM-like RISC processor to support application tasks that are not computationally intensive. These processors would be distributed over the platform tiles shown in Figure 3.2, e.g. the MPEG2 decoder on tile 13 could be a VLIW DSP, while the MPEG2 encoder on tile 6 could be CGRA. The camera and LCD display interfaces are also present in this SoC platform. A large amount of on-chip data memory is distributed on the platform to reduce off-chip communications and to allow the storage of several MPEG2 frames on-chip. For the platform energy breakdown presented in this section, only one processor type is used to serve as an example, namely the VLIW DSP. A detailed description and comparison of the other above listed compute node styles is presented in the second part of this case study (see Section 3.3).

Tile sizing and communication In this experiment, it is assumed that the platform consists of 16 large heterogeneous tiles, contrary to some other research that considers very small tiles [Wie02], resulting in hundreds of tiles on a chip. This choice is motivated by the fact that distributing heavily communicating sub-tasks over several tiles leads to a high energy consumption over the global inter-tile communication network. That is especially true in scaled technologies where wires start to fully dominate the power/energy consumption in a design. Instead, we opt for larger tiles consisting of a compute node and its local memory. This leads to a second type of communication, the intra-tile communication. If the applications are split in such a way that the intra-tile communication is completely design-time analyzable, it can be heavily optimized and is less expensive than the inter-tile communication,

which requires run-time hardware support [Guo08]. The intra-tile communication typically consists of the communication between the processor and the local memories on one hand and the communication between different slots on the other hand. The communication with the local memories can be kept small by optimizing the local lay-out (see also Section 3.6) and the types of processors that are considered here, except for the CGRA, have a limited amount of local interconnect. Therefore, this local communication is not considered further in this chapter. For RISC and VLIW processors, the cost of this local interconnect is here included into the energy cost of the communicating components. For CGRAs, this assumption does no longer hold and this is handled in detail in Chapter 4 of [Lam09].

The tiles in this experiment are sized in such a way that they are able to execute complete complex tasks (e.g. MPEG encoding). These tasks are still only parts of the complete applications (in this case, the video encoder/decoder chain), but are split in such a way that most of the communication remains intra-tile. The platform tiles also include a reasonable amount of embedded L1 caches or scratchpads, which requires a careful design of cache and scratchpad size (for a target application or application domain) [Das05]. In this way, a well-optimized tile mapping can help to reduce the performance degradation and power consumption due to inter-tile communication overhead.

Moving the first level of cache or scratch-pad inside the tile, together with the assumption that the application has been heavily DTSE-optimized, leads to a heavily reduced energy consumption for the inter-tile communication architecture (as will be shown by the experimental results of this section).

Tile placement In order to correctly estimate the energy consumed by the inter-tile communication, the placement of the different tiles onto the chip and the respective wire-lengths have to be known. In this experiment, we have chosen to distribute the compute nodes over the platform with a good average placement relative to memories, as opposed to a placement that is optimized for only certain tiles or applications. Using more specific task load information, the placement can be further optimized for a narrow application domain by clustering heavily communicating tiles. More information on this type of inter-tile communication optimization can be found in [Guo08].

3.2.1.3 Inter-tile communication architecture

Shared buses are still a popular choice for on-chip communication, but they do not offer the scalability and extended bandwidth that is required for future multimedia platforms. Buses connecting a large amount of tiles quickly

become a bottleneck and are not energy-efficient as the whole bus has to be driven each time an access is performed. Networks-on-Chip (NoC) have been proposed as an alternative to buses [Dal01], next to other more scalable bus-variants, like bridged buses or sectioned buses [Pap06]. As the optimization and exploration of Communication Architectures is not the topic of this book, a NoC-based communication has been chosen, because of its ease of use, which makes it a strong possibility for future platforms.

In a NoC-based platform, each tile is connected to the network through its own network interface, which is connected to a router (a group of tiles can also share a router). Routers are linked to a sub-set of the other routers to form a specific topology (e.g. a mesh). A routing algorithm determines which path must be taken from source to destination.

For this experiment, a regular mesh has been chosen as the logical view of the network, because it heavily simplifies the task of the network control algorithm and allows the network to be reconfigured fast enough for very dynamic applications like 3D graphics. The network uses switched virtual-circuits, a technique that consists of establishing an exclusive connection from source to destination. Part of the network resources and bandwidth are thus dedicated exclusively to this connection, as long as it is not released. To find and reserve the best circuit of a given bandwidth across the network from source to destination, an adaptive routing algorithm is used that allows to efficiently avoid paths that are entirely occupied by other existing circuits [Shi03].

3.2.1.4 Mapping the application to the architecture

The logical view of the platform (Figure 3.2) shows the mapping of the application. A set of assumptions have been made about other applications running simultaneously on the same platform, in order to achieve a globally more realistic view of the platform, as the current application alone would never generate any congestion or routing problems. As the communication network is not the focus of this book, the reader is referred to [Ler08] for more information about these assumptions. The resulting mapping is described here briefly. The camera and display interfaces (I/O) are placed at the border of the chip and do not require a large area (tiles 1 and 7 respectively). The MPEG2 encoder and decoder are mapped respectively on a coarse-grained architecture (tile 6) and on a VLIW (tile 13). The encoder and decoder working memories are chosen (from the available memories) to be as close as possible to these processing nodes. The dark arrows show the paths that are set up by routing algorithm of the virtual circuit switched network. In this example, it is assumed that the maximal bandwidth is reached on some links (indicated by a cross) and as a result, some communication can not follow the shortest route (e.g. from MPEG2 decoder to Display).

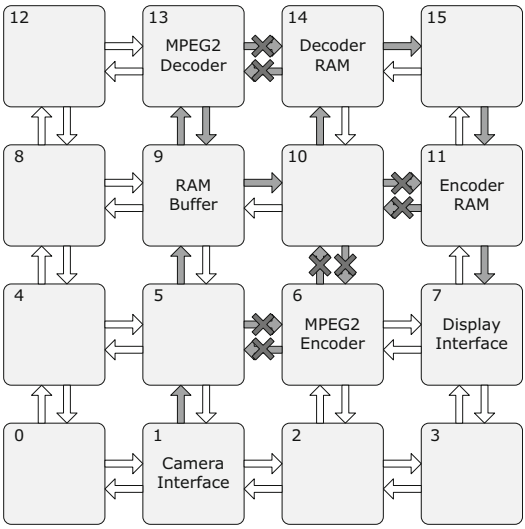


Figure 3.2: Mapping of the application on the logical platform view

Other application tasks may run simultaneously on the unoccupied nodes of the platform (e.g. a 3D-graphics chain that will produce a smaller picture-in-picture game window on the shared display). The data and instruction memory hierarchies are based on state-of-the-art clustered/distributed organizations at the local level and a shared SDRAM at the top level.

A more detailed description of the NoC modeling, a detailed description of the routers and the control algorithm to set-up the virtual circuits can be found in [Ler08].

3.2.2 Experimental results for platform components

This section presents the experimental procedure, the energy and power estimation and the contribution of different platform components.

3.2.2.1 Experimental procedure

The case study that is presented in this chapter combines the results of several experiments, each modeling a certain aspect of the platform. The reported results for the different levels of the memory hierarchy and for the datapath operations correspond to one particular compute node, namely the MPEG2 decoder mapped on a homogeneous VLIW processor with eight slots

and have been generated using the COFFEE platform instruction-set simulator that is based partly on the older CRISP simulator (see Chapter 4). The processor types described in Section 3.2.1.2, will all be compared in the second part of this case study (Section 3.3) for the MPEG2 decoder, in order to compare their respective performance, while normally the different tasks described here would be mapped on different processor types.

The reported power consumption of the inter-tile communication architecture corresponds to the cost for all communication in the whole platform, based on computed band-widths for the different links. In order to be able to estimate these bandwidths, DTSE optimized has been assumed, such that the transferred amount of data could be estimated from the size of a frame (assuming two reads and one write). The energy estimation was performed for the individual components of the NoC, but no network simulator has been used for the complete described NoC. Still, we believe this provides a reasonable estimate that is at most a small factor below the actual value. And even with a larger error offset, the conclusions that are discussed in Section 3.2.3, will not change due to the relatively small contribution of the NoC in the entire power pie, due to the energy-efficient mapping (especially because of applying the DTSE stage) of the network.

3.2.2.2 Embedded processor datapath logic

The embedded processor used in this section is a VLIW DSP, running the MPEG2 decoder. A comparison with other processor styles is presented in Section 3.3. For the platform energy breakdown of this part, a *centralized heterogeneous* VLIW architecture is used, as is shown in Figure 3.3. If all slots in a VLIW contain the same FUs and hence can execute the complete processor instruction (operation) set, the VLIW is called *homogeneous*. If the slots are specialized and can execute only a subset of the supported operations, as is the case for most commercially available VLIW architectures [TI04], the VLIW is called *heterogeneous*. When all slots share the same register file, the register file is called *centralized*, while the processor is called a *clustered* VLIW if the slots are grouped and each group has its own register file. In this experiment, a centralized register file of 32 entries is used. Clustered VLIW architectures are discussed in Section 3.3.

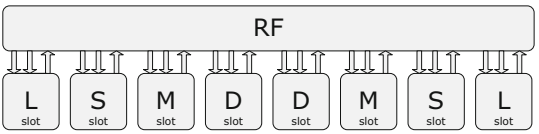


Figure 3.3: Heterogeneous centralized VLIW processor architecture

The different slots of Figure 3.3 are modeled after the TI-C6x-family of heterogeneous VLIW DSPs. Four different types of slots, called L, S, M and D, each support a sub-set of the processor instruction set. Their functionality is as follows:

- L: Arithmetic, Compare and Logical operations
- S: Arithmetic, Shift, Logical and Branch operations
- D: Arithmetic and Load/Store operations
- M: Multiplications

The embedded processor has been simulated using the the COFFEE platform instruction-set simulator (see Chapter 4). The MPEG2 decoder application has been taken from the $Mediabench$ [MedB] benchmark suite. This task decodes a sequence of four 4-CIF frames (IPPP). The processor, a heterogeneous VLIW, has eight slots of which the supported operations for each slot are modeled after the TI TMS320C6204 [TI04]. The energy cost per operation for the datapath assumes the use of optimized module generator (semi-custom) data reported by [Fei03] for a processor implemented in a 0.13 µm CMOS technology node and running at 1.2 V. The instructions that are executed on the VLIW are categorized as ALU operations on one side and MUL operations on the other side. This assumption has been used as an approximation, since in most processors the difference in energy consumption of most non-MUL operations does not vary significantly [Fei03]. This is partly due to the control overhead and the overhead of the fetch and decode stages. Therefore, Arithmetic and Logic operations, Loads and Stores and Shift operations are all grouped into the ALU group, which has a significantly lower power consumption than that of the operations in the MUL group. For scaled architectures, we expect the cost of (larger) multiplier to increase relative to the cost of the ALU. The significantly higher cost of MUL operations is further discussed in Chapter 10.

Table 3.1 presents the energy and power consumption estimates for the processor datapath separately.

3.2.2.3 Datapath pipeline registers

The energy cost of the pipeline registers is normally included in the energy cost for the datapath operations, which leads to higher numbers as reported by others. In this case, we have consciously made a split between the datapath logic and the datapath pipeline registers to take out the effect of increasing the clock frequency on the energy consumption of the core. In this

	MUL ops.	ALU ops.	Total
Number of ops.	7,883,491	366,006,736	373,890,227
Energy cost per op.	4.86E-12 J	2.24E-12 J	
Total Energy	3.83E-5 J	8.19E-4 J	8.58E-4 J
Power at 25 fps			5.36 mW

Table 3.1: Energy and power estimation for a module-generated (semi-custom) datapath logic, for the decoding of four MPEG2 frames at 25 fps

experiment, the energy that is spent in pipeline registers has been modeled and computed separately, as the core clock-frequency is assumed to be quite high (600 MHz).

For a target frequency of 600 MHz, we assume that a pipeline depth of 12 stages is required (which is in line with the pipeline depth of the Ti C62 family). Our computations show that a 32-bit pipeline for eight parallel slots consumes a total of 82.9 mW at 0.13 µm and 1.2 V. In contrast, the same datapath will require only four pipeline stages and consumes only 9.2 mW if that processor is running at 200 MHz (assuming no Vdd scaling has been applied). An increase with a factor 3 in frequency leads to an increase in the pipeline energy consumption of about a factor 9.

3.2.2.4 Data and instruction memory hierarchy

Foreground data memory The foreground memory, the memory closest to the datapath, is considered to be part of the processor (the core). In the case of a VLIW processor, the foreground memory is the register file. A centralized register file in a 8-slot VLIW processor needs 16 read ports and eight write ports. It has been modeled using energy per access estimates obtained from [Ben01]. The register file has 32 entries and consumes 98 mW. Table 3.2 clearly shows that the power consumption of the register file is an important part of the power breakdown. This power consumption can be reduced by moving to a clustered register file, which will be discussed in Section 3.3.

Background memory and loop buffer The background memory hierarchy typically consists of a level 1 instruction cache (IL1, 8kB), a level 1 data cache (DL1, 8kB) and a unified level 2 (UL2, 256kB) cache (both for instructions and data). With just a single level of instruction cache, we observed that the instruction memory hierarchy formed the largest bottleneck. Therefore, we decided to introduce a loop buffer (can be seen as the foreground instruction memory, but is only useful for loops) as presented in [Jay02a]: this small

Architecture component	Power consumption (mW)
Data path logic	5.4
Pipeline registers	82.9
Centralized register file	98.2
Loop buffer	84.6
Level 1 instruction cache	25.9
Level 1 data cache	134.8
Unified level 2 cache (DTSE optimized)	3.2
Communication architecture (DTSE optimized)	7.1

Table 3.2: Power breakdown for the embedded platform, running a video encoder/decoder application. The results for the processor components and memories are for the MPEG2 decoder application only, while the communication architecture includes the cost of all traffic of the encoder/decoder application

register file can store the instructions for a loop and has a lower energy cost per access than the level 1 cache. During loop execution, the datapath gets the instructions directly from the loop buffer that can store 64 instructions and the IL1 is not accessed (can be switched to a low power state). During non-loop code, the processer accesses the IL1 directly. This approach reduces the number of accesses to the larger and more expensive instruction cache, which results in a reduction in the power consumption. For more information on this optimization, the reader is referred to [Jay02a].

The higher levels of instruction and data caches were modeled using cache models obtained from [Log04]. The loop buffer has been modeled as a single ported register file.

This experiment has shown a power consumption of 84.6 mW for the loop buffer, 25.9 mW for the L1 instruction cache, 134.8 mW for the L1 data cache and 3.2 mW for the unified L2 instruction-data cache.

3.2.2.5 Inter-tile communication architecture

As the physical implementation of the virtual mesh is no longer a regular mesh, because the different tiles of the platform contain processors of different types and memories of different sizes, the tile size and link length are very heterogeneous. Therefore, a high level layout has been used using realistic area estimates for the different tiles, to estimate the link lengths (Figure 3.4).

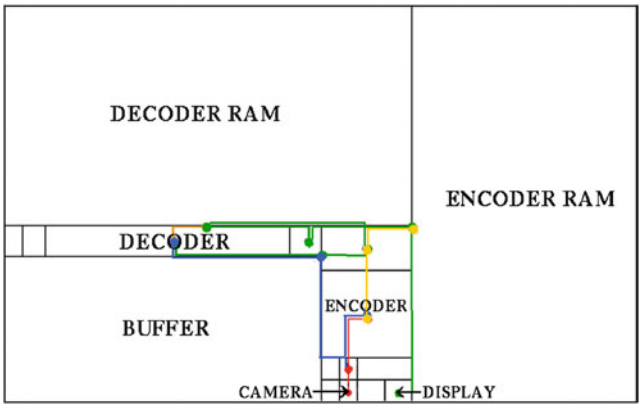


Figure 3.4: Physical view of the platform (die size = 140 mm^2 for a 0.13 µm technology node)

The power consumption of the virtual circuit-switched network has been estimated for a network composed of 16 routers interconnected to the nodes through network interfaces. A detailed energy estimation has been performed for a 0.13 µm implementation, based on a gate level simulation, using real traffic. More details on the router power consumption estimation approach can be found in [Lam05] and [Ler08].

The inter-tile communication architecture contributes a total power consumption of 7.1 mW for executing the 25 fps video chain. This cost is rather low, as a result of the above mentioned decisions on tile sizing and the fact that the accesses to the background memory are DTSE optimized. As a result, this cost can be considered to be a lower bound on the global communication but even with a significant increase (e.g. by a factor 2) due to less design effort being spent, the inter-tile communication architecture will still not be a dominant component in the overall power pie.

3.2.3 Power breakdown analysis

The power consumption for the different components of Table 3.2 is presented as a pie-chart (Figure 3.5), which gives the relative contributions of the different parts of the platform. As has been mentioned before, in order to draw correct conclusions from this breakdown, the analysis of the relative order should include the assumptions that were used to conduct this experiment. In the following paragraphs, important assumptions and their consequences will be discussed in detail and conclusions will be drawn.

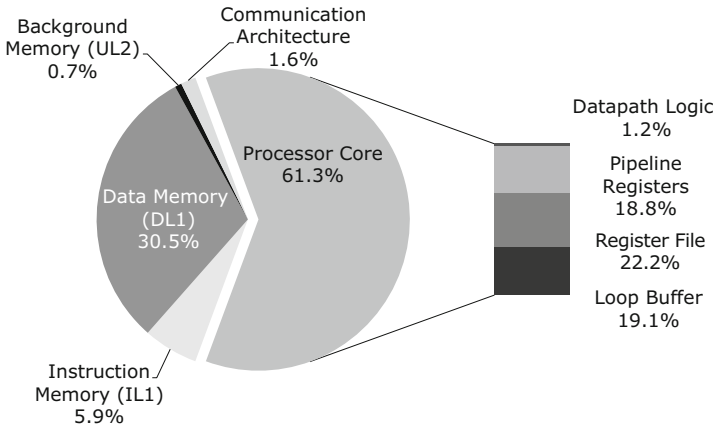


Figure 3.5: Power breakdown for the embedded platform, with the components of the processor core grouped

Datapath logic and pipeline registers For the datapath logic, the energy per activation costs are based on module generator-based (semi-custom) functional units that have been reported in [Fei03]. These have been used instead of estimates for a standard cell based design which is more and more dominated by wiring overhead. As many industrial designs still use standard cell datapaths, the energy cost for the datapath logic can increase up to around 5% or more for those conventional systems.

When more performance is needed and the processor will be running at higher frequencies (e.g. up to 1 GHz), the number of pipeline stages will increase heavily, which leads to a severe increase in the energy and power consumption of the datapath. This can lead to an energy consumption in the pipeline register of up to 30% overall [Rag08b]. On the other hand, when more parallel architectures that can deliver the same amount of performance at a lower clock frequency (e.g. down to 200 MHz) are used, the pipeline depth can be reduced and the energy spent in pipeline registers will be reduced heavily. At the same time, the number of operations will increase (exploiting more parallelism mostly leads to a certain number of $extra$ operations, e.g. pack/unpack, predication as a form of speculative execution etc.), which will lead to a larger contribution of the datapath logic. Combined with a standard cell technology, the datapath logic will become an important part of the breakdown. An example of this trend can be seen in Section 3.3 and in the energy breakdown for the CGRA in Chapter 4 of [Lam09].

Overall, we can conclude that the estimated cost for the datapath logic in this experiment can be seen as a lower bound and that the energy consumption in

most state-of-the-art processors today will be significantly larger. The energy consumed in pipeline registers is significant and varies with the processor clock frequency.

Foreground data memory A large portion of the pie chart of Figure 3.5 is the foreground memory, here the register file. The design of the register file is a crucial factor, which can dominate the total system power consumption.

The number of read and write ports to the register file is a critical factor that determines the register file power consumption. Based on the parallelism available in the application and the performance requirements, the designer must try to reduce the number of read and write ports, which is linked to the design of the datapath. The use of hierarchical register files or clustered register files can partly reduce the number of ports and thereby the cost per access of the register file. A reduction of 40% for the power consumption of a two-level hierarchical register file (HRF), compared to a single centralized register file, is reported by [Zal00a]. A comparison with clustered register files is presented in Section 3.3.

As the register file is an important contributor to the total cost, this component will be discussed in more detail in Section 3.3.

Data memory hierarchy The power consumption of the data memory hierarchy represents a significant amount of the pie-chart. Programmers often write applications in a way that separates production and consumption of data using temporary storage, because this is easy to understand. However, this leads to many unnecessary copies and extra accesses to the data memory. These unnecessary accesses can be removed by techniques like DTSE, or by other data management techniques, using code transformations [Moo97]. In this experiment the accesses to the level 1 data cache contain such redundant accesses and therefore can be significantly reduced. As no multi-level cache simulator was available, the accesses to the background memory (level 2 cache) and the corresponding transfers over the inter-tile communication architecture have been estimated based on the required amount of data that needs to be minimally transferred per frame, starting from the frame size. This assumption, specifically two reads and one write per frame, holds for DTSE-optimized code, when data re-use is maximized and additional accesses are minimized. The use of simulated accesses for the level 1 cache with non optimized code and estimated accesses for the background memory has to be taken into account when interpreting these results. As the expected improvements of applying the full DTSE methodology are a reduction with a factor 2–5 [Cat98b, Cat02], a corresponding increase in accesses to the Background Memory can be extrapolated. However, this will still not increase the power consumption of this component to a point where it would

become a bottleneck. In contrast, applying DTSE to the DL1 data memory, which currently is a large consumer (30.5% in Figure 3.5), is needed and we assume that this would reduce the cost for this component significantly (at least with a factor 2 [Cat02]) and it will no longer be a bottleneck.

Additionally, circuit level techniques, like drowsy caches [Fla02, Kim04], selective activation of the caches, efficient bank organization etc. may be used to reduce the static energy and the cost per activation. Given the large amount of research spent on optimizing the data memory hierarchy, we assume that the overall cost of this part can be heavily reduced with respect to the cost shown in the breakdown of Figure 3.5. However, it is still one of the major consumers and it is important to make sure that processor optimizations do not add additional cost to the data memory. The DTSE methodology inherently ensures this by an appropriate phase ordering in the different steps and by propagating the necessary constraints from each step to the next one [Cat98b, Cat02]. In this way, the processor mapping decisions cannot become overly constrained and the decisions in that later stage cannot undo the obtained cost reductions in the DTSE stage.

Instruction memory organization As parallel architectures, like VLIW processors, execute multiple operations per cycle, they require a wide access to the instruction memory per cycle, which can be very costly. As a unified level 1 instruction memory would fully dominate the power consumption in such a case, a loop buffer has been used in this experiment. However, the combined power consumption of the IL1 and the loop buffer is still consuming 25% in Figure 3.5. To further reduce the power bottleneck in the instruction memory, techniques like clustered or distributed loop buffering, source code transformations [Pet03] and instruction compression [Kad02] can be introduced, in addition to the simple loop buffering scheme that has been used in the here presented experiment. As with the data memory hierarchy, we acknowledge that the instruction memory organization is a large consumer of energy, but we assume that the energy consumption can be reduced with respect to the cost shown in Figure 3.5 by implementing the above listed optimizations (e.g. [Baj97] reports up to 60% reduction when using a loop buffer and [Jay02a, Jay05b] reports an additional 60% for clustered, distributed loop buffers). Again, it is important to verify that processor optimizations do not have effects on the instruction memory that would move the overall energy consumption to a globally worse point.

Inter-tile communication architecture Contrary to common belief, the power breakdown shows that the global communication architecture is not dominant for well chosen tile sizes and a well-optimized mapping, at least in our target domain of handheld multimedia terminals. Some assumptions are at the basis of this (to some) surprising conclusion. By choosing tile sizes

that are larger than many tiles reported in literature, a part of the communication has been pushed into the tiles, namely the communication between the processor slots and the foreground memory and between the processor and the first level data cache. As this communication is guaranteed to be very local, the energy spent there is assumed to be small and is neglected here. It should be noted that the floor-planning of the actual nodes in the network leads to a very heterogeneous physical network model with different lengths and bandwidth requirements for the links, which should be taken into account by a lay-out-aware routing algorithm to minimize the cost. If this is not done, the communication cost would increase due to sub-optimal routing decisions (see motivation in [Ler08, Ler06b, Mur09]). The reported estimate is assuming that DTSE has been performed to minimize the global communication (and the accesses to the background memory). If this is not the case, the traffic can be as much as five times more (depending on missed re-use opportunities and redundant transfers), leading to a corresponding increase of the energy spent in the inter-tile communication.

We can conclude that energy spent in an optimized communication architecture can be low, although the estimate reported in this chapter should be considered to be an optimistic lower bound. As the communication cost does not dominate the total system at this stage, we do not focus on communication any further.

3.2.4 Conclusions for the platform case study

It can be clearly seen in Figure 3.5 that the processor (datapath + foreground memory), the data memory hierarchy and instruction memory hierarchy consume equally important parts of the total platform power. The absolute cost of the datapath (Functional Units performing the real computations) is quite low, but, as the processor style (see Section 3.3) heavily influences the cost of foreground memory and the instruction memory hierarchy, the importance of the processor is still high. Therefore, a large need exists to optimize processor, instruction and data memory hierarchies, keeping track of their effect on each other. If it is optimized like in this experiment, the energy or power consumption of the inter-tile communication architecture can be kept sufficiently low.

3.3 Embedded processor case study

Major innovative solutions are needed to merge energy efficiency, flexibility and high performance into a single embedded processor. A first step in this direction is the evolution from RISC to VLIW. VLIW processors

provide more computing resources, and rely heavily on the compiler to use these resources efficiently. VLIWs thereby avoid the energy-hungry hardware resources (e.g. dispatch unit) that are extensively used in super-scalar processors. Because even higher performance and higher energy efficiency is needed, different, mostly domain specific VLIW-descendants are currently being developed. However, no clear overview of these different design styles and their advantages and disadvantages was available in public literature at the time of our publication [Lam05]. To be fair and really valid this overview needs to be based on concrete data for the same realistic application, that is separately optimized for the different styles. In the context of the case study that has been presented in the previous section, a comparison of different processor styles and their performance and energy consumption is presented for the MPEG2 decoder application. In contrast to the results presented in Section 3.2, all results presented in this section are generated in the same way and therefore can be directly compared.

Recently, this study has been extended in [Por06], adding more processor styles and substantiating the conclusions of this work.

3.3.1 Scope of the case study

The processor styles compared here include both traditional instruction set processors (*software style*) and *reconfigurable hardware*. The performance (cycle and operation count) of different processor cores is estimated running the MPEG2 decoder part of the presented video processing chain.

We assume that when techniques like DTSE [Moo97] are used, the energy consumption of the data memory hierarchy will be relatively low compared to the other unoptimized parts of the platform, as described in Section 3.2.2. Moreover, the optimized organization can be assumed to be quite similar (certainly in terms of energy consumption) for all covered processor styles, because the same application is mapped with the same throughput requirements. So, except for the local variables in the foreground data memory (e.g. register files), which will vary heavily in the different styles, the accesses and the related energy consumption to the higher levels of the memory hierarchy will be very comparable. Because of this, the source code transformations to better exploit the data memory hierarchy are not discussed any further in this work. In Chapter 7 we will describe though how to handle irregular indexed arrays and dynamically accessed data structures on our platform, in order to show the wide use domain of our approach.

When the data memory hierarchy is optimized, the instruction memory hierarchy will consume a significant part of the total energy [Vij03]. Moreover, the instruction memory hierarchy exploration is coupled to the architecture

style of the compute node/processor, as increasing the number of parallel resources leads to an increase in the number of instructions to be fetched every cycle. In this section, the results of an instruction memory hierarchy exploration that is described in [Vda05], will be summarized. Apart from the summary, the optimization of the instruction memory hierarchy is outside the scope of this book. However, because of its closer link with the processor style, effects of processor optimizations on the instruction memory cost have to be closely monitored in order to reduce the overall system cost.

The Instruction Memory Hierarchy is also called Configuration Memory for the so called reconfigurable hardware processors. Both components are very similar and different proposed optimizations can be fully shared over the two target styles. Therefore, the used terminology is from here on shared and we will call it the Instruction or Configuration Memory Organization or ICMO from now on.

3.3.2 Processor styles

The computation intensive nature of future embedded applications has moved the designers' choice from sequential RISC processors towards more parallel architectures, like the VLIW (see Figure 3.3). In a VLIW, multiple (typically four to eight) slots operate in parallel. A 4-slot VLIW can therefore issue four new instructions per cycle. The VLIW paradigm is relying on the compiler to extract sufficient Instruction Level Parallelism (ILP) to keep all slots busy. Every slot contains one or more Functional Units (FU) that operate mutually exclusively and execute the operations that are allocated on that slot. A slot can typically execute a range of similar operations, e.g. an Arithmetic and Logical Unit or ALU executes arithmetic operations like additions and subtractions, but also logical operations (AND, OR etc.) and relational operations (equal to, greater than etc.).

Because a VLIW compiler decides which instructions can be executed in parallel, the processor does not need hardware that performs this job at runtime, as in the case of a super-scalar processor. This leads to a more energy efficient, less complex processor, in which more resources can be dedicated to do the actual computations.

The conventional heterogeneous centralized VLIW architecture, as was used in Section 3.2 and is called *VLIW baseline* further on, relies on mature compiler support, but suffers from two major drawbacks. Firstly, the multi-port centralized data register file (with three ports per slot) and the very wide instruction memory hierarchy (all slots need to access the ICMO every cycle) of the baseline consume too much energy. Secondly, the baseline VLIW architecture is not scalable to higher performances (up to tens of slots) as is required by future multimedia applications. These applications contain very regular

and computation intensive loops (kernels) that can provide more parallel operations than a VLIW processor with four to eight slots can exploit. The quest to minimize the energy consumption for a given performance, as required by the application, has lead to the development of a whole range of VLIW "descendants", with several types of extensions, both *compiler techniques* and *architectural optimizations*. This section will summarize some approaches that try to improve the energy efficiency, the performance or both.

3.3.2.1 Software pipelining (e.g. modulo scheduling)

Goal: performance improvement and indirectly (instruction + datapath) energy reduction A VLIW processor can only achieve the promised speed-up over a RISC (only one slot) and energy reduction over a superscalar processor (no hardware to detect parallel operations) if the compiler succeeds in keeping the extra resources busy as often as possible. Traditionally, VLIW compilers have concentrated on finding independent instructions in the sequential instruction stream. However, reported data show that compilers, on average, can keep only three to five slots busy by using only Instruction Level Parallelism or ILP [Mei03b]. Modulo Scheduling (MS) is a software pipelining technique used by the compiler to extract more parallelism by executing multiple iterations of a loop at the same time. This type of parallelism (Loop Level Parallelism or LLP) is abundantly available in multimedia applications. The most compute intensive parts of these applications, called kernels, consist mostly of nested loops. MS allows different loop iterations to be executed in an overlapping manner, and can keep up to tens of slots busy during the execution of the kernels, depending on the data dependencies that exist between successive loop iterations. This results in a higher performance if the architecture has sufficient slots to exploit this kind of parallelism. As the available hardware will be more efficiently used if more slots are filled per cycle, which is expressed as the Instruction Per Cycle count or IPC, the energy efficiency of the instruction memory and the datapath will also improve. An alternative way to exploit the application parallelism is SIMD or sub-word parallelism, which is discussed later in this section.

For very regular applications that can be software pipelined, it can happen that more parallelism is available than can be exploited by a traditional VLIW.

3.3.2.2 Clustering Clustering ($clustered\ VLIW$)

Goal: energy efficiency improvement The energy consumption of the multi-ported centralized data register file of the baseline can be reduced by grouping the slots in separate clusters. By using a separate, smaller register

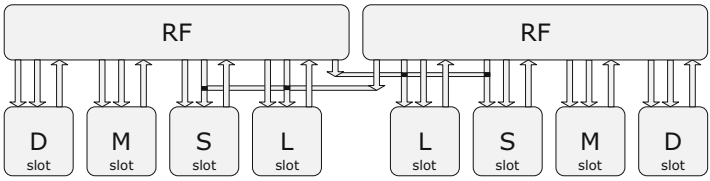


Figure 3.6: Clustered VLIW with two clusters of four slots each, modeled after the TI-C6x-family

file with fewer ports for every cluster, the total area and energy consumption of the register files can be reduced. Figure 3.6 shows an example of a clustered VLIW modeled after the TI-C6x-family. In this case, the L and S slots also support a cluster copy operation, which reads a value from the register file of the other cluster and stores it in the local register file. This inter-cluster-copy operation consumes a cycle and only one value per cycle can be read from or written to the other cluster.

Clustering complicates the compilation, as the compiler needs to take the restrictions in the communication between clusters into account. When the clusters become too small, the number of extra copies that are needed to communicate between clusters will start to have a negative influence on performance. Other ways to implement inter-cluster communication exist (e.g. a sub-set of the units can directly execute on data that is read from the other cluster or the use of forwarding paths). In any case, due to the extra restrictions that the compiler has to take into account, increasing the clustering for an equal amount of slots leads to a performance degradation. This type of clustering can also be successfully applied to the instruction hierarchy, namely to the loop buffers.

3.3.2.3 Coarse-grained reconfigurable architecture

Goal: performance improvement and potentially also good for energy efficiency A second possible way to increase the performance of the baseline VLIW processor is by adding extra resources, e.g. adding more (rows of) slots (Figure 3.7). One slot, a base unit of the CGRA, contains a set of mutually exclusive FUs (as in VLIWs), but also contains local routing resources (MUXes) and potentially local register files or part of the distributed loop buffer, which is here called the configuration memory. This base unit is called a Processing Element or PE and the complete architecture is called a Coarse-Grained Reconfigurable Architecture (CGRA).

The spatial ordering of the PEs can be in the form of a matrix, in which case this type of processor is also called a Coarse-Grained Reconfigurable Array.

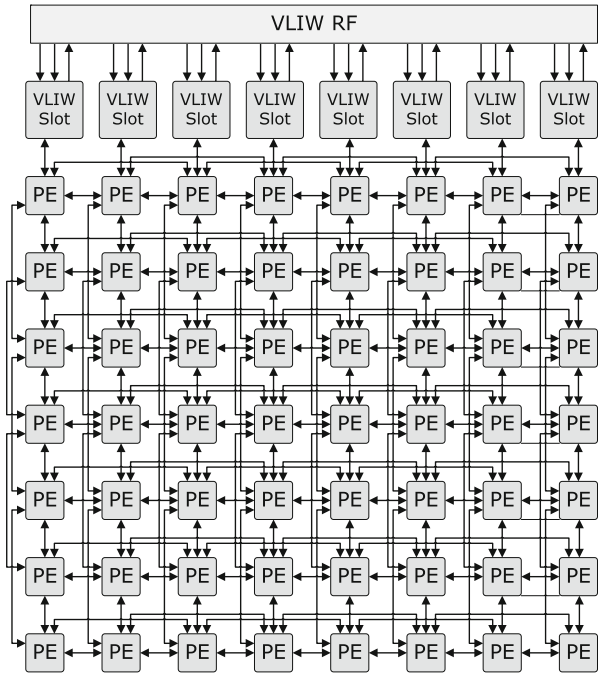


Figure 3.7: The ADRES CGRA, with tightly coupled VLIW processor

However, the interconnect can be organized differently, leading to hierarchically or more one-dimensionally organized structures, which are conceptually all part of the same CGRA family. A compiler for this architecture needs to be able to exploit the highly parallel architecture in order to boost the performance, but also faces the extra task of routing the data through the array.

Figure 3.7 shows one example of a CGRA, in which the first row of the array literally is a VLIW processor of the same width as the reconfigurable array. This is not necessarily the case, as the VLIW can be narrower or can be removed completely. More details on this specific CGRA architecture called ADRES (a CGRA developed at IMEC vzw [IMEC]) can be found in [Mei03a].

The CGRA can be seen as a second baseline style, but can also be considered as a VLIW extension [Mei03a]. Due to the large amount of interconnect in a CGRA, the ISS-based energy estimation approach that is used for the VLIW architectures is not usable. For a solution to this problem and a detailed energy breakdown of the more complex CGRA processor, we refer to Chapter 4 of [Lam09].

3.3.2.4 SIMD or sub-word parallelism

Goal: performance improvement and potentially a large impact on energy Another way to be able to execute more operations in parallel is to use a sub-word parallel, vector or SIMD approach. Instead of adding extra slots, the FUs can be made to operate on different data in parallel (Data Level Parallelism or DLP). This means that a 32-bit wide FU can, e.g. do a single 32-bit, two 16-bit or four 8-bit operations. The operations that are performed in parallel are of the same type, e.g. all additions, hence Single Instruction Multiple Data or SIMD. Next to the performance improvement, the energy consumption can be reduced because almost no extra operations are needed if the mapping is performed effectively [OdB03].

In multimedia or wireless communication applications, word sizes of 8 and 16 bits are very common. Again the success of this approach depends on how well the compiler, in this case called a vectorizing compiler, can take advantage of the sub-word parallel capabilities of the hardware. As the compilers that are used in this book do not provide automatic vectorization, all sub-word parallel code has been parallelized manually. Due to the large potential performance improvements, manually parallelizing important kernels is common practice.

A discussion on the trade-offs with respect to exploiting LLP, ILP and DLP, see Section 3.6.5.

3.3.2.5 Custom instructions and/or FUs

Goal: performance and energy efficiency improvement If a VLIW is targeted towards a certain application or application domain, specialized instructions can be added to the instruction set. These custom instructions combine a number of frequently used operations into one instruction and are executed on specialized FUs that are more efficient (in time and energy). This technique can have a very big positive influence on performance and energy consumption if the target application domain effectively makes use of the new instructions and FUs. The disadvantage is the area overhead.

On the other hand, if only limited flexibility is needed, the number of supported instructions can be reduced to support only a small set of applications. This customization leads to a cost reduction for the ICMO (less bits are needed) and the datapath itself (less area and a reduced energy consumption).

The disadvantage of both approaches is a loss of generality/flexibility (best suited for an ASIP).

3.3.2.6 Optimized data memory hierarchy

Goal: energy efficiency improvement and performance improvement by reducing stalls By adding extra layers to the data/instruction memory hierarchy, locality of data/instructions can be exploited. Data or instructions can be moved to small memories, closer to where they are needed and can be accessed with a smaller delay and lower energy penalty. This results in better performance and less energy consumption. More information about options and trade-offs will be provided in Section 3.6.2.

3.3.2.7 Hybrid combinations

It should be stressed that the actual processor data-paths proposed in the literature are usually hybrid combinations of the above extensions. It indeed makes much sense to combine them partly for addressing the requirements of a given application domain. Good recent examples in the commercial domain are the TI OMAP [TI09c] and the ARM Cortex [Bar05b]. But many other hybrids have been proposed in the academic literature in the last decade. Also our own template proposal in Section 3.7 is essentially a well-chosen hybrid that is optimized for our ultra low-energy target domain.

3.3.3 Focus of the experiments

Most of the concepts that have been presented in the previous paragraphs can be considered to be decoupled. This means that a specific processor design can implement combinations of these approaches, of which only some influence each other. The concepts can be categorized in two different ways. On one hand, a distinction can be made between techniques improving the energy efficiency and those improving performance, as has been already indicated by the *Goals* in the individual paragraphs. On the other hand, these concepts can be separated into extensions of the baseline styles and supporting techniques. In this book, (1) the *clustered VLIW*, (2) the *sub-word parallel VLIW*, (3) the *VLIW with custom FUs* and (4) the *coarse-grained architecture* are considered to be extensions of the VLIW baseline design style. They modify the baseline architecture and can be considered descendants of the *centralized VLIW*. Software pipelining and the usage of a memory hierarchy can be applied to all these variants and are considered to be supporting techniques.

In this section, the performance of the different styles is compared: a RISC (separate baseline style, used as a reference), a centralized VLIW, a clustered VLIW and a CGRA (with closely coupled VLIW). A comparison with an

FPGA-soft-core, and ASIC and a specialized ASIP was recently added by [Por06]. The goal is to get a feel of the differences between these styles and their effect on performance and energy consumption. The sub-word parallel VLIW and the custom instruction approach are not included in this first exploratory comparison, but these techniques are used for experiments in later chapters (see e.g. Section 9.4.4 for a comparison on different SIMD implementations and Section 10.3.4 for custom instructions). To make a good comparison, also the supporting techniques are used. Both the compilers of the clustered VLIW and the VLIW with closely coupled coarse-grained architecture use modulo scheduling to extract sufficient parallelism to efficiently fill these architectures (achieve a high utilization).

An important part of the total energy consumption of embedded systems is consumed by the ICMO. The choice of the processor design style has a strong influence on this part of the system. Moving from RISC processors to more parallel architectures, like VLIWs and coarse-grained architectures, means fetching more instructions in parallel. In these parallel processors, architecture-specific new techniques can be used to reduce the energy consumption of the ICMO. A separate study of ICMO extensions has been conducted as part of this case study. Different implementations of loop buffers have been compared. Optimizations at micro-architectural, compiler and software levels have been proposed to improve the energy efficiency of the instruction memory, using the same tool flow, energy models and driver application. The results show that the energy consumption of the instruction memory for different VLIW architectures (centralized and different clustered versions) can be reduced by up to 60% depending on the implemented loop buffer variant. More information about this ICMO optimization can be found in [Vda05, Lam05].

3.3.4 Experimental results for the processor case study

The results for the different processor styles that will be presented in the following section are for the MPEG2 decoder, taken from the Mediabench benchmark suite. The results show the required number of operations and cycles to decode a 4 frame MPEG2 reference sequence (IPPP) with a frame size of 4-CIF. These performance estimates are produced by the respective tool flows for the different processor styles, using optimized code, the processor-style specific compiler and cycle-accurate simulator and including the effects of the memory hierarchy, like load/store latency and stalls (for all processors but the ADRES CGRA). For comparison, the number of decoded frames per second is shown for all processors running at 600 MHz. The results are presented in Table 3.3.

	# ops. ($\times 10^6$)	# cycles ($\times 10^6$)	IPC	fr./s at 600 MHz	ms/fr.
(a)	243	336	0.7	6.9	145
(b)	252	86/60	2.9/4.0	27.9/39.9	36/25
(c)	311	95/72	3.2/4.3	25.2/33.3	40/30
(d)	337	84/64	3.9/5.2	28.5/37.5	35/27
(e)	*331*	*26*	*9.3*	*92.4*	*11*

(a) RISC (ARM920T), (b) centralized VLIW (eight slots), (c) clustered VLIW (2×4 slots), (d) clustered VLIW (3×4 slots), (e) *ADRES* (8×8 *PEs*)

Table 3.3: Performance simulation results for different processor design styles: all reported results are for Impact semantics. The re-computations from Elcor to Impact semantics mention worst case/best case assumptions (separated by a /). The ADRES CGRA result does not include the effects of a memory hierarchy

3.3.4.1 RISC

As a first reference case, the decoding of the four frames has been simulated with ARMulator for an ARM 920T embedded RISC core. This resulted in a total of 243 million operations, and 336 million cycles. This means that at 600 MHz, this processor can decode almost 6.9 frames per second. The number of executed Instructions Per Cycle (IPC) in this case is 0.7. This is clearly far from the performance that is needed in this context (15–30 frames per second). Even for very high clock frequencies, this issue would not be solved yet.

3.3.4.2 Centralized VLIW

For the second reference case, the VLIW baseline, the heterogeneous centralized VLIW with eight slots (as in the first part of this case study) has been simulated using the COFFEE processor architecture simulation and exploration environment [OdB01, Rag08b]. This retargetable research compiler and simulation framework for VLIW-like architectures was built on top of Trimaran [Tri99]. The framework uses an Impact front-end for standard architecture independent compiler optimizations and the Elcor back-end for target specific register allocation, scheduling and assignment. Elcor has been originally developed for the HP PlayDoh architecture [Kat00], but is customizable to other targets using a machine file. The COFFEE simulator reports the total number of cycles and the total number of operations needed to execute the MPEG2 decoder for four frames. It also reports the amount of reads and writes to different levels of the memory hierarchy.

Consistency between different frameworks To be able to correctly compare the produced results, in number of operations and cycles, the detailed instruction set of the processors had to be compared. Because of differences in the underlying assumptions with respect to the target instruction set of the Impact back-end (for the CGRA), and the Elcor back-end (for the VLIWs), the produced results are not directly comparable. Elcor decomposes some instructions (post-increment/decrement, branch/prepare-to-branch etc.) into multiple operations, while Impact does not. To make a fair comparison possible, results produced by the Trimaran framework have been recomputed into Impact semantics by removing the extra operations. As a simple removal in some cases will lead to the reduction of the critical path, a shorter schedule can be obtained. Additionally, the IPC is reduced because operations are removed, but potentially increased because of a shorter schedule. Because the automatic estimation of the Trimaran flow was not usable for the modified generated schedule (removing the extra operations leads to an invalid Trimaran schedule), two re-computations are presented. The optimistic re-computation assumes that the IPC will stay the same for the reduced number of operations and re-computes the cycle count as $\#ops./IPC$. This estimate is reported as "best case". The pessimistic, or "worst case" result assumes that only cycles that are completely empty (NOP on all slots) after the extra operations have been removed, can be removed from the cycle count. The lower IPC is then calculated based on this lower cycle count as $\#ops/\#cycles$. The recomputed results are mentioned between brackets (worst and best case separated by /) and are summarized in Table 3.3.

Tweaking the compiler settings The results of this experiment are highly influenced by the performance of the used compiler flow, in this case COFFEE: Impact/Elcor. The first straightforward result, obtained by compiling the unoptimized Mediabench MPEG2 decoder code, has resulted in a cycle count of 484 million and an operation count of 824 million, or an IPC of only 1.7. This shows that on average less than two of the eight slots are used. This very bad performance, even worse than the RISC, was the result of a mismatch between the compiler front-end, Impact, and the Elcor back-end of Trimaran. An aggressive loop unrolling in Impact prevented Elcor from performing modulo scheduling in the kernels and generated a lot of spill code. This explains the extra operations and a degraded performance. It shows that simply using public domain compiler software to compare different solutions does not guarantee valid conclusions. A large effort had to be spent to find out all the issues involved and to modify the compilation setting accordingly to obtain a correct comparison.

Changing the parameter settings of the front-end, turning off the default aggressive unrolling, had the biggest impact. By doing this, the result improved

to 182 million cycles and 392 million operations, or an IPC of 2.1. This corresponds to decoding 13.2 frames per second at 600 MHz, which is still not impressive.

Local platform-dependent source code transformations Another important issue that is often overlooked in comparisons is the impact of source code transformations. The way the code was initially written, heavily influenced the outcome of the experiments. By performing very processor specific manual optimizations to the MPEG2 decoder code, the performance has been increased to 88 (86/60) million cycles and 373 (252) million operations, or an IPC of 4.1 (2.9/4.0). In this case, on average, more than half of the slots are occupied every cycle. At 600 MHz, this processor can decode 27 (27.9/39.9) frames per second. The manual optimization consists of well known code transformations (e.g. loop merging, loop unrolling, etc.) that have to be applied with the specific architecture in mind. To obtain a fair comparison, a significant effort was spent to create optimized versions for all the investigated styles. This experiment shows that mapping an application to a certain platform is highly influenced by the quality of the compiler. Large improvements can still be made by (up to now) manual source code transformations.

3.3.4.3 Clustered VLIW

The COFFEE simulation framework extends Trimaran by adding support for data-path clustering. This includes (1) assigning operations to a data-path cluster, (2) scheduling inside the clusters (assigning operations to a certain slot, in a certain cycle), using modulo scheduling for the kernels, and (3) adding necessary copy operations to communicate between clusters.

Effect of clustering on performance A clustered VLIW with two clusters of four slots was modeled to compare to the eight slot centralized VLIW of the previous paragraph. The simulations show that the clustered processor needs 95 (95/72) million cycles and 415 (311) million operations, which corresponds to an IPC of 4.3 (3.2/4.3). This means it can decode 25 (25.2/33.3) frames per second. The decreased performance (8% more cycles needed) and the significant increase in number of operations (12% more) are the result of the communication overhead between different clusters. The compiler has to add extra inter cluster copies. In this case, looking at the overall IPC only, would lead to the wrong conclusion. In fact, less instructions that belong to the original algorithm are executed per cycle.

A clustered VLIW with three clusters, each containing four slots, has been included in this experiment to see how performance scales with respect to an

increase in resources. The decoding takes 84 million cycles and 445 (337) million operations, with an IPC of 5.2 (3.9/5.2). The number of frames that can be decoded at 600 MHz would be 28 (28.5/37.5). In this case, the very small performance improvement can not justify the increase in resources by adding an extra cluster. The higher operation count will lead to a higher energy consumption for the same task. The positive effect on performance of adding more resources (more clusters) is quickly countered by the performance degradation of the communication overhead between the clusters. Even though the IPC can still increase, the amount of $useful$ operations per cycle often does not, as an increasing number of cluster copy operations are required. The number of clusters that can efficiently be used, depends on the amount of parallelism available in the application, and on the ability of the compiler to exploit it.

Effect of clustering on the energy consumption As has been explained in Section 3.3.2.2, the goal of clustering is to save energy. Because the energy consumption of a centralized register file quickly dominates the energy budget of a processor as the number of ports grows [Zyu98], clustered register files allow designers to reduce the energy consumption of their VLIWs or to keep the energy consumption of large VLIW processors under control.

As mentioned above, VLIW processors exploit their extra resources to increase performance, but the cost of multi-ported register files will cause an important energy penalty. Table 3.4 presents the energy consumption of the register file for the different VLIW configurations under study. The power model used is linear with respect to the number of simultaneous read/write accesses and to the number of read and write ports, as reported in [Ben02].

Compared to the centralized VLIW with eight slots, the clustered VLIW (2×4 slots) can bring the energy consumption of the VLIW register file down with almost 80% (from 98 to 22 mW), at the cost of a small decrease in performance (a reduction in frame rate of two frames per second). At the

	#Reg file clusters	#read ports (/cluster)	#write ports (/cluster)	Power (mW)	Energy (mJ)
(a)	1	16	8	98	13.3
(b)	2	8	4	22	11.1
(c)	3	8	4	26	12.5

(a) Centralized VLIW (eight slots), (b) clustered VLIW (2×4 slots), (c) clustered VLIW (3×4 slots)

Table 3.4: Register file energy (for four simulated frames) and power consumption (scaled to 25 frames per second) for the different clustered versions of a VLIW processor

beginning, the performance of a clustered VLIW increases as more resources are added. After a certain point (in this case already for 3×4 slots), the performance improvement reduces due to the inter-cluster communication and the inability of the compiler to use all the clusters. Due to the extra operations, the power consumption can again increase significantly. In this experiment, an increase in the energy consumption of 18% has been shown for the register file of the 3×4 slot VLIW (from 22 to 26 mW), compared tot the clustered register file of the 2×4 slot VLIW, while the performance increase is only three frames per second.

3.3.4.4 Coarse-grained architectures

Experiments for the coarse-grained architecture have been performed with the retargetable DRESC compiler [Mei02], which combines a modulo scheduler, register allocator and 2D router. This compiler can map complete applications written in C to the ADRES architecture [Mei03a]: a coarse-grained array with tightly coupled VLIW processor. DRESC uses modulo scheduling to map the regular and compute intensive kernels of the application on the CGRA, while more control dominated parts are mapped to the VLIW part of the architecture. The DRESC compiler re-uses the Impact research compiler (front-end and back-end) to compile to this VLIW part. More information on ADRES/DRESC can be found in [Mei02] and Chapter 4 of [Lam09].

The simulated architecture consists of an 8×8 array of PEs, of which the first row is a normal VLIW. The other PEs are ALU-like resources or multipliers, with support for predication and routing capabilities. The reconfigurable interconnect architecture allows results from an operation performed on one PE to be consumed by other PEs in the architecture without passing through the centralized register file. The DRESC simulator does not include a cache simulator, and assumes a fixed latency for the data and instruction memory hierarchy. Therefore, the results for this case do not include the performance effects of realistic memory hierarchy. A realistic implementation would add extra stalls because of cache misses. Because of this, the reported figures are optimistic with respect to the delivered performance.

The decoding of the four MPEG2 frames on ADRES takes 26 million cycles and 331 million operations, or an IPC between 15 and 35 for kernels that are mapped to the CGRA and an overall IPC (including the parts that are executed on the VLIW) of 9.33. Excluding the predicated (nullified) and the routing operations, the overall IPC is 8.8. Running at 600 MHz, this architecture would be able to decode 90 frames per second.

The CGRA successfully boosts the performance and can decode up to 90 frames per second at 600 MHz, which means that a lower clock frequency can be chosen, which could be beneficial for the energy consumption. However,

the number of operations has again increased. These extra operations partially needed to route data between PEs, but the ability of the CGRA to directly move data between PEs reduces the number of accesses to the register files. A second source of extra operations is the use of predication to allow the mapping of kernels that contain a limited amount of control-flow onto the CGRA.

Because of the high number of resources, the area of a CGRA is significantly larger than the other options. It should also be noted that the performance increase for an increase in resources is gradually reducing (over a certain threshold). A detailed energy breakdown of a CGRA architecture is the subject of Chapter 4 of [Lam09].

3.3.5 Conclusions for the processor case study

The design style case study for embedded processors/compute nodes shows that improvements for energy efficiency and/or performance over currently used RISC or VLIW processors can be achieved.

The results presented in Table 3.3 show that a RISC processor can do the job with the smallest number of operations. However, it is clear that a RISC, even for higher clock frequencies, does not provide the performance that is needed for this real-time multimedia application. A centralized VLIW processor exploits its extra resources to increase performance, but the energy cost of a centralized register file will be prohibitive to integrate this type of processor into platforms with stringent energy constraints. A clustered VLIW can bring the energy consumption down, at the cost of a (small) decrease in performance. After a certain point, however, the performance deteriorates due to the inter-cluster communication and the inability of the compiler to use all the clusters. The CGRA with tightly coupled VLIW can really boost performance for applications that are regular enough to keep the vast amount of resources busy (sufficient loop level parallelism with limited control flow). However, the price that has to be paid is an increase in the number of operations, which is expected to increase the energy cost for the same task. The energy analysis of a CGRA is presented in Chapter 4 of [Lam09].

3.4 High level architecture requirements

Figure 3.8 shows the different design styles and the key metrics on which they are efficient. ASICs are known for their high performance and energy efficiency. DSPs on the other hand are flexible as well as deliver the performance, but they are not energy efficient. Application Specific Instruction-set

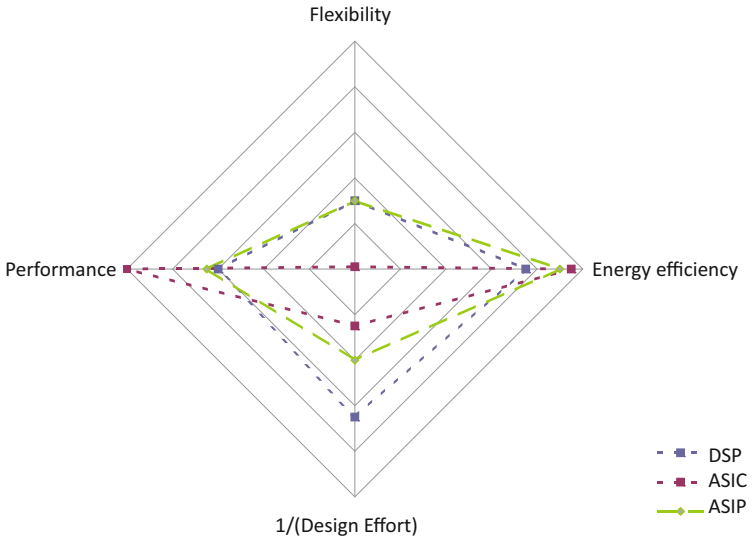


Figure 3.8: Different design styles target different design metrics

Processors (ASIPs) on the other hand try to combine all three metrics. ASIPs try to reach the same energy efficiency and performance of an ASIC while still being flexible. To design such an efficient ASIP or embedded processor, it is necessary to observe the high level requirements and trends in the domain of processor design. However the design effort for mapping code on an ASIP is higher.

The goal of this book is to bring the ASIP based programmable solution as close as possible to the ASIC based solution in terms of performance as well as energy efficiency. This chapter presents a high level analysis of the space of processor architectures in an embedded platform. Also such a high level and global analysis would ensure that the different parts of architecture and the compiler are consistent with each other and each of the optimization does not destroy the optimization of another step. Such a high level analysis of trends and requirements is required before finding the solutions for each of the individual parts and to reach an energy-efficient ASIP solution.

While this book focuses on some parts of the processor, it impacts all parts of the platform. To ensure this work is consistent with optimizations in other parts, it is required to look at all parts of the platform. A common analysis and trade-off discussion of proposed architecture extensions ensures the consistency between the different parts. The different contributors have focused on different parts, but have also taken the effect of local modifications on the rest of the platform into account.

The result of this common analysis is presented in this chapter and together all proposed modifications form the FEENECS architecture template (Section 3.7). By explicitly looking at the requirements and restrictions of current compilers and their link with the architecture, high-level relations have been derived in order to improve the compilability. Based on these relations and on the specific features of the FEENECS architecture template, a matching compiler proposal has been drafted (see [Lam09], [Rag09b] and the patent filing [Rag08a]).

3.5 Architecture exploration and trends

Architectures form the bridge between the application and the technology. Therefore to optimize an ASIP processor architecture, the designer must take into account the application requirements. An effective ASIP architecture exploration has to cover a wide range of architectures to find the one which is Pareto optimal for the application and system cost trade-offs (e.g. reduce the energy consumption while providing the required real-time constraints and quality). From the implementation side, it is important to take the physical design method (e.g. custom design vs. standard cell design) and high level technology inputs (e.g. poor interconnect scaling in Deep Sub-Micron or DSM technologies, leakage) into account early in the design flow to ensure a more optimal implementation. If the implementation allows the designer to give guidelines on the floor-plan, it is important to take this into account. Architecture exploration therefore forms the corner stone of any processor design.

Note that the variability and reliability impact on processor platforms is a crucial issue in DSM technologies and they are the subject of much recent research (see, e.g. recent proceedings of the DATE, DAC, HPCA conferences). But the mitigation of these effects can be handled in a complementary way [Wan07, Wan09] that is compatible with our overall platform template proposal and mapping approach. These aspects will however not be tackled in this book.

3.5.1 Interconnect scaling in future technologies

A number of papers have appeared that compare the scaling of interconnect to the scaling of logic (transistors) for scaled technology. According to many papers [DeM05, Jos06, Syl99] and the latest ITRS report [ITR07] a clear difference exists between logic and interconnect scaling and, interconnect scales much worse. This leads to potentially reduced gains when scaling to Deep Sub-Micron (DSM) technologies ($65, 45, 32, 22$ nm). The poor scaling

of interconnect and vias is due to varying physics limited factors ranging from the k-value between the wires as well as due to DSM issues like surface scattering and grain boundary scattering. This affects both local and global wiring and therefore this trend needs to be taking into account for future architectures.

In this chapter, it is assumed that interconnect does indeed scale worse than logic and that the impact of this interconnect scaling in terms of performance and energy cost is increasing. Therefore, some rather disruptive modifications to the architecture are proposed (e.g. replacement of traditional register files with new foreground memory structure, as discussed in Section 3.6.3). The architecture modifications that are proposed in this chapter are propagating the higher cost of interconnect through to various levels of the design (from heavily communicating components, based on application knowledge, down to layout).

However, in case the cost of interconnect (especially the local interconnect) does not increase with respect to logic, the gains of the proposed solutions will still exist with respect to traditional designs, but may relatively reduce. Therefore some of the traditional state-of-the-art solutions will still be part of the valid trade-off solution to choose from. It may also be the case that because a disruptive change often takes more effort, the current state-of-the-art architecture may still be taken.

3.5.2 Representative architecture exploration examples: What are the bottlenecks?

During and after architecture exploration, the designer can obtain an energy breakdown of the different components of the processor architecture. Figure 3.9 shows the energy break-down for a high-performance CGRA processor implemented in 130 nm running a MIMO application. Figure 3.10 shows the energy break-down for an embedded VLIW processor running the MPEG2 decoder.

While both breakdowns are from different application and processors, similar conclusions can be drawn. From both figures, it can be seen that the energy consumption is not really dominated by a single architectural component.

In the CGRA a large part of the energy consumption is spent on the interconnect (the architecture instance that is shown is a high-performance instance with a rich interconnect topology). This is because a significant part of the communication between different slots in the VLIW has been pushed from the VLIW register file to the interconnect and the PE Pipeline Registers. This leads to a relatively smaller part of the CGRA energy that is directly consumed by shared register files. In fact, in a CGRA the foreground memory organization

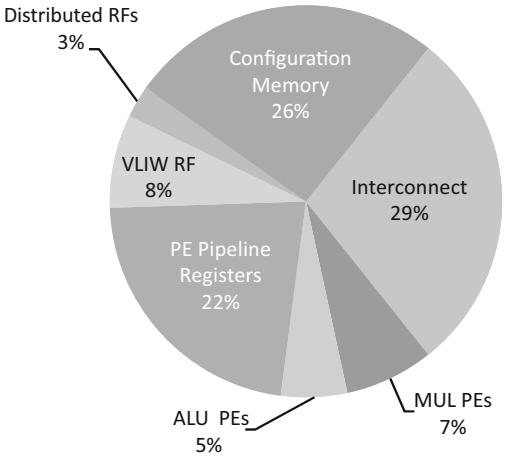


Figure 3.9: Energy break-down for a high performance CGRA (8×8 PEs) running a MIMO benchmark, with a clock of 200 MHz

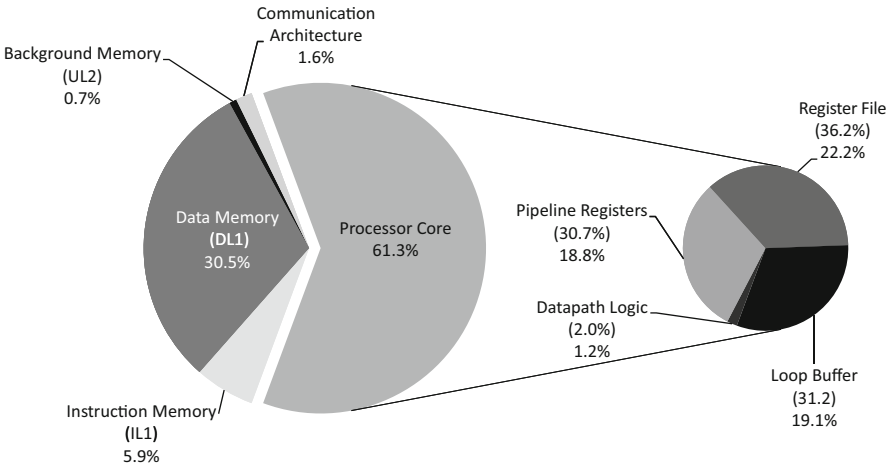


Figure 3.10: Energy break-down for heterogeneous VLIW processor with eight slots running an MPEG2 decoder, with a clock of 600 MHz. The numbers between brackets indicate the percentage for the processor core only

consists of the register files, pipeline registers and the interconnect and therefore still remains a critical issue.

It is assumed here that in order to reach the required high performance, the eight slot VLIW has to be clocked at a significantly larger clock speed (here 600 MHz compared to 200 MHz for the CGRA).

The CGRA is more parallel than conventional VLIWs and a larger part of the energy consumption is directly spent on the datapath logic (in this case 12% for both ALU and MUL PEs, compared to only 2% for the VLIW), which indicates a more efficient energy usage. Note that neither of the pie-charts gives direct information about the difference in absolute energy consumption between the VLIW and the CGRA.

In conclusion, both pie-charts show that the configuration memory or loop buffer, register files, datapath logic, datapath pipeline registers and interconnect consume almost equally important parts of the global energy. Therefore, all parts need to be considered together while optimizing.

3.6 Architecture optimization of different platform components

A clear need exists to think globally and across different abstraction layers while designing an architecture. The properties required at each of the architecture components and the reasons for these properties can be motivated across the different abstractions layers. This cross-abstraction knowledge can lead to different optimizations than would be selected if every part of the system optimized in isolation, which we call *entanglement*. The following subsections present a possible solution for individual architectural components and its reasoning across different abstraction layers.

3.6.1 Algorithm design

Even though this section targets architecture optimizations, the cross-abstraction information can be propagated down from the algorithm design. Following the increase in processor parallelism and the inability to exploit irregularity in algorithms (with many conditions), algorithmic designers have been pushed to design more regular algorithms. In case of power efficient ASIC design, once again designers are forced to use regular algorithms as adding flexibility is expensive in ASIC design. However, when these trends leads to a drastic increase in the number of operations that need to be executed, the energy penalty is significant (e.g. compute motion compensation in MPEG as a full search or hierarchical search). Even though the parallel architecture can be better filled and performance can be better, the resulting efficiency in terms of energy per task is still (significantly) lower. Therefore, some of the modifications that are proposed in this section explicitly address the need to support more useful irregularity in an efficient way, directly on the parallel architecture. These modifications include a split in address and

data computations onto separate slots (Section 3.6.6), distributed control loop buffers (Section 3.6.4), and efficient parallelization of irregular data (Section 3.5 in [Rag09b]). When the architecture supports more irregularity, different versions of an algorithm can be designed, taking into account the context. One promising example with large gains (up to a factor 10 in energy efficiency and performance) can be found in [Li09], where the conditions of the wireless channel are taken into account in order to select different implementations of an FFT algorithm to match the requirements instead of executing a full FFT in all cases.

The rest of this section will focus on the different architecture components and discuss the propagation of constraints from the application down to the layout.

3.6.2 Data memory hierarchy

Application As the amount of data required increases substantially from one generation of the application to the next, e.g. a higher data rate in wireless applications or a higher resolution in image/video applications, it is important to efficiently handle the transfers of this large amount of data. Typical embedded applications exhibit both spatial and temporal data locality, which can be exploited (using source code transformations as discussed in [Bro00a]) to reduce the cost of the data memory hierarchy by optimizing the reuse.

Architecture When the application can be analyzed at compile time, the data transfers can be managed by the programmer or compiler and a scratchpad can be used instead of a cache [Ste02, Ban02, Pan98, Mar03]. In the case of a scratchpad, the data transfers from the higher level memories to the scratchpad memory are programmed explicitly and handled by a DMA engine. This programming overhead is acceptable as it can be done at design-time. In rare cases where embedded applications exhibit truly random accesses and their access pattern can not be analyzed at compile time can still use (hardware controlled) caches [Abs07].

Implementation From the perspective of layout and circuit design it is not always possible to design one large memory. Because of the increasing cost of long interconnect [Jos06, Syl99, ITR07], very large monolithic memories are increasingly difficult to build in scaled technologies. Therefore the memory can be partitioned into banks and internally into sub-banks. Furthermore the use of multiple levels of hierarchy and the DMAs should also be taken into account while floor planning.

3.6.3 Foreground memory organization

In case of VLIWs, register files form the core part of the communication of data across the different slots and clusters as a storage element. Alternatively, as in CGRAs, part of that communication can be pushed to the interconnect between different PEs. Therefore the centralized and distributed register files, the PE pipeline registers and the inter-slot or inter-PE interconnect are all considered to be part of one architectural component and they need to be optimized together. From here on register files and the connections between the slots are together called the *foreground memory organization* in the rest of the book.

As *foreground memory* forms one of the core parts of the processor architecture and given that it is one of the biggest bottlenecks, new ways to optimize them needs to be considered.

Application *Different types of data:* Current embedded application contain various types of data, from array data that have high spatial locality to scalar data that store, e.g. a single coefficient or a temporary value. Unlike the data layout of the data memory hierarchy, that explicitly allocates (parts of) arrays to scratchpads and optimizes the transfers with local copies, at the register file all data is treated the same. Array data is stored together with temporary variables, without any concept of data layout. By introducing the data layout concept at the level of the traditional register file, the specific properties of different data elements in terms of spatial locality, temporal locality, size, life time etc. can be exploited. In conventional architectures and compilers this is not yet done.

Architecture *Heterogeneous register file:* To be able to split the different types of data and treat them accordingly, a heterogeneous foreground memory architecture is needed. A separate register file for scalar variables will enable to perform a more optimized data layout for the streaming data (e.g. a wider register file for SIMD data and a narrower scalar register file). Splitting off the address computations into separate slots (see below) enables the use of an optimized register file for the address path.

Architecture: *Energy cost per access:* From the architectural perspective, about 50% of the power consumption of a typical L1 data memory (around 64K) is spent in the decoding logic. The other 50% is spent in the actual data storage, in the memory cells [Amr00, Eva95]. The spatial locality that is available in array data of the streaming type can be exploited by loading them

together into the register file by a wide load/store from the memory. The overhead of the address decoding in the memory will thereby be distributed over a number of words.

Additionally, the energy cost per access scales with the number of ports [Rix00a], as an increase in the number of ports leads to an increased load for every register file cell. Therefore multi-ported register files can be replaced by a clustered register file architecture, with multiple register files that have less ports each. From the cost per access point of view, a form using only single-ported register file cells is therefore advisable. In today's architectures this is not achievable due to restrictions in both architecture and compiler, that would lead to poor performance and utilization. Therefore a more disruptive foreground memory change is needed to reach this goal, where architecture and compiler modifications are coupled/entangled.

Implementation Finally, from the layout perspective it is important that the wire lengths of the interconnect inside the register file and of the complete foreground memory organization (including the connections to the other slots) should be optimized. Accesses to the foreground memory are very frequent and heavily communicating components are close together. This activity-aware floor planning is compatible with the approach presented by [Guo08], but here the concept is applied at a finer granularity.

Proposed solution Some intermediate values in parallel architectures often have a single cycle life-time, it does not make sense to store the variable into the register file organization. In this case it is more efficient to use the forwarding network, that writes the result of one Functional Unit after the execute pipeline stage directly into the pipeline register before the execute stage of the next FU. This type of forwarding is commonly found in DSPs and forms an integral part of the proposed foreground memory organization.

For variables with a longer life-time, a novel architecture solution is proposed. Figure 3.11a shows a typical clustered register file where the interfaces between the memory and the register file on one hand and between the datapath and the register file on the other hand are of equal width. Data can be copied word by word into the register file, with no restrictions on the data layout in the L1 data memory. Figure 3.11b shows the proposal of a Very Wide Register (VWR), a foreground memory organization optimized for streaming applications that exhibit a large amount of spatial locality. The VWR has a wide interface (width of a complete SPM row) towards the memory and a narrow interface (width of a datapath word) towards the datapath. From the datapath side, through an interconnection network, SIMD FUs can read out data words that internally contain many sub-words. The wide

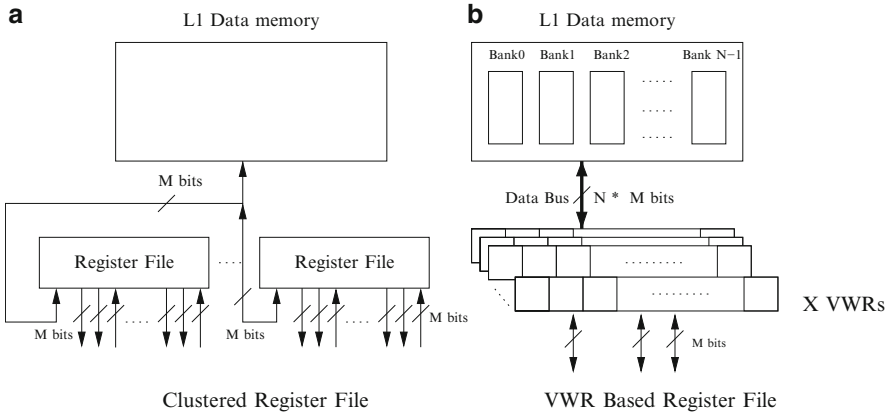


Figure 3.11: Very Wide Register: A register file solution for streaming data with spatial locality

interface towards the memory allows a large amount of data to be transferred to/from the VWR with a single memory decode. This implies a reduced decode overhead for the L1 memory, but requires a careful data layout and increased compiler complexity to maximize the number of useful data words to be present in this transfer. One multi-ported register file can be replaced by a set of VWRs. While there are effectively two ports one towards the memory and one towards the datapath, the VWR storage cells remain single ported as both the ports are not simultaneously accessed. This single ported cell nature of the VWR internally reduces the energy cost per access compared to a multi ported register file, at the cost of an increased complexity for the compiler. The physical layout of the VWR and SPM can be matched (pitch alignment) to reduce the interconnect length between both. Therefore the layout of the physical implementation should be such that the VWR and SPM are placed next to each other.

The gains that can be expected when moving from a traditional register file architecture to a VWR-based foreground memory organization depend on the assumption that the cost of interconnect increases when scaling to DSM technologies and the improvements will be less if the cost of interconnect can be reduced by technology modifications.

The VWR forms a key architecture extension discussed in this book. A more detailed description of the VWR, its micro-architecture, its operation and the potential gains is detailed in Chapter 8. The compilation technique for a VWR based architecture is detailed in [Rag09b].

3.6.4 Instruction/Configuration Memory Organization (ICMO)

A traditional instruction memory organization consists of an L1 instruction memory which is controlled by a program counter (PC). As increasingly parallel architectures require more instructions to be fetched every cycle, the instruction memory needs to be wider and therefore consumes a lot of energy.

Application Applications typically consist of different control flows merged into a single combined control flow, e.g. the address-generation part of the program is mixed with different data producer–consumer chains into one sequence of operations. Mixing the different flows seemingly simplifies the programming, but the efficiency of the instruction memory can be improved if the different flows can be handle separately. To enable this in current architectures is however not possible.

Additionally, most applications contain both control intensive parts and regular kernels that perform the most computationally intensive parts. The kernels are structured as nested loops and form the core of most embedded applications. By matching the instruction memory organization to the kernel structure, the overall efficiency can be improved. Many applications also contain kernel with incompatible loop nests that are potentially executed concurrently in the final task schedule.

Architecture Traditional instruction memory organizations consist of a monolithic L1 instruction memory, controlled by a single program counter (PC). Recently academic and some industrial architectures have introduced a small instruction memory closer to the processor, called *loop buffer* or *L0 memory* [Sia01, IBM05, Uh99]. This small L0 memory contains the instructions for a kernel, together with a small zero-overhead loop (ZOL) control or a loop controller (LC). During loop execution, instructions are fetched from the loop buffer and the L1 instruction memory can put to sleep (by either Vdd throttling or clock gating). Due to the extra copy that is needed to move the kernel instructions from the L1 to the loop buffers, there is a trade-off involved. For some instructions in control-intensive parts of the code it will be difficult to gain back the cost of this extra copy. Therefore these control-intensive parts are still directly fetched from the L1. This approach is compatible with the use of non-volatile memories for the L1 instruction memory in order to reduce leakage energy consumption in the L1 memory.

The concept of a loop buffer can not only be used for different slots of the datapath, but is also useful for other platform components that need to be programmed, e.g. the DMA, interconnect networks, etc. However, the activity

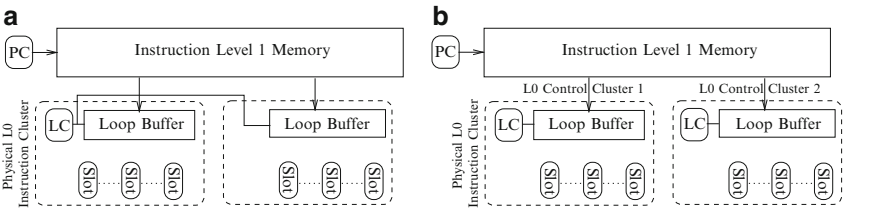
of e.g. the DMA and the datapath can be very different. Therefore sharing a single loop buffer and keeping the control centralized will normally lead to an inefficiency.

To obtain a more optimal solution the control of loop buffers for different parts of the architecture can be split in order to match the application constraints, resulting in *clustered and distributed control loop buffers*. By splitting the control, components that are not active for a certain number of cycles can be kept under low-leakage/sleep mode. These distributed loop buffers can also have a local controller per partition [Rag09c]. That allows to support a form of software-controlled multi-threading. Moreover, the different loop nests can now be incompatible which allows the concurrent execution of distinct loop nests and required by many modern applications. And finally each of the partitions can now be separately optimized for size and access count, enabling the better exploitation of spatial and temporal locality.

Implementation *Physically distributed loop buffers* are placed close (during layout) to the parts of the processor they control. This can be achieved using activity aware floor-planning techniques like [Guo08] and distributing the loop buffers as explained in [Jay05a].

Figure 3.12a shows a state-of-the-art physically distributed loop buffer as explained in [Jay05a]. Figure 3.12b shows the proposed physically distributed loop buffer with distributed control which would enable different parts of the program to be controlled in an efficient way, based on its corresponding activity of that part.

The distributed control based loop buffer is a key architecture extension in this book. A more detailed description of the clustered respectively distributed control and its operation can be found in Chapters 5 and 6.



VLIW processor physically distributed loop buffers with a centralized single loop controller

VLIW processor withphysically distributed loop buffers and distributed loop controllers

Figure 3.12: Clustered and distributed loop buffer: An instruction memory solution for optimal instruction issuing

3.6.5 Datapath parallelism

Although the datapath operations are not a bottleneck in Figures 3.9 and 3.10, the datapath style has a large effect on how the register files and instruction/configuration memory organization will be used. The organization of the datapath decides on the complexity and consequently on the cost of the interconnect. Current embedded processors exploit parallelism in order to provide a sufficient amount of performance, while still keeping the energy consumption under control. They can however differ in the way they do this. As not all approaches are as energy efficient, this section discusses the trade-offs involved.

Application The computationally intensive kernels of most embedded applications contain parallelism at different levels: e.g. across different pixels, blocks of pixels or frames of a video sequence. Different types of parallelism follow from the way it is extracted from the application. When different iterations of a loop are being executed in parallel, this is called Loop Level Parallelism (LLP). The parallel execution of different instructions, be it outside or inside a loop is called Instruction Level Parallelism (ILP). Finally, the execution of multiple instructions of the same type on different data is called SIMD or Data Level Parallelism or DLP. The amount of parallelism of different types that can be extracted depends on the application dependencies. Ideally, processors should contain a mixture of slots of different widths: very wide SIMD units for regular kernels that contain a large amount of DLP, together with medium wide units for kernels with more control and scalar units for non-DLP code.

Architecture Embedded processors can be designed to exploit one or a combination of these types of parallelism. Both embedded VLIW processors and CGRAs provide parallel slots or PEs and use software pipelining to convert the LLP into ILP (see Figure 3.13). *Separate instructions* (although combined into a single very large instruction in VLIW terminology) control the parallel slots or PEs, which gives a lot of freedom to the compiler with respect to how to overlap the different iterations (still respecting the dependencies) and how to place the different instructions on the slots. As a downside, exploiting ILP requires managing a large number of separate units and the communication between them for every instruction of the overlapped loops, which leads to a significant amount of energy that is spent in the ICMO (Instruction/Configuration Memory Organization).

When subsequent iterations can be overlapped completely, the parallelized operations are of the *same type* and subword parallelism or SIMD can be used. This means that they can be executed on the same FU (wider and

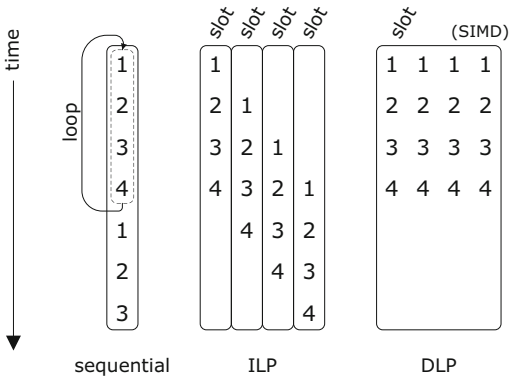


Figure 3.13: Converting Loop Level Parallelism (LLP) into Instruction Level Parallelism (ILP) or Data Level Parallelism (DLP)

slightly modified to separate the data). The advantage of exploiting DLP is that the resulting SIMD instruction executes multiple (e.g. 4 as shown in Figure 3.13) operations in parallel, but is still steered by a single instruction, hence Single Instruction Multiple Data or SIMD. This leads to a more energy efficient parallelization than exploiting ILP.

A common approach to extract DLP from a loop is to overlap multiple iterations of a loop. The amount of DLP that can be extracted this way corresponds to the number of iterations that can be fully parallelized without breaking dependencies between the iterations. This freedom is usually limited.

Architectures can use exploit both ILP and DLP. State-of-the-art embedded VLIW processors support SIMD [TI00, VdW05], while subword parallelism in CGRAs is far less common.

In conclusion, the total number of separate slots can be reduced for the same performance by using DLP. Therefore the relative contribution of the cost of the instruction/configuration memory to the total energy consumption can be reduced. Secondly, the reduction in the number of slots leads to a reduction in the complexity of the interconnect. Because of this motivation, the processor architecture should first exploit all available DLP and only then, if required, a limited amount of ILP can be used to reach the required real-time performance. The ICMO should also be organized in a distributed fashion and customized to the different slots to be efficient.

Implementation As with the register file architecture and physically distributed loop buffers, it is important to place the datapath slots as close as possible to where they are needed. For example the units that will compute

the addresses (Address Generation Units or AGUs) can be placed closer to the interface to the memory and do not need to be grouped with the data computation units (see next section). Additionally, the most heavily communicating slots can be placed next to each-other in the floor plan. In order to optimize the implementation of the datapath units, without going to a full custom design, so-called Datapath Generators (DPG) can be used. Given that it is more favorable to exploit SIMD, each of the sub-word datapaths can be optimized well using semi-custom logic instead of random standard-cell based place and route. Such a semi-custom/custom based datapath will consume significantly less area and energy, as is demonstrated in [Wie01].

3.6.6 Datapath–address path

Application Operations that contribute to the execution of the target application are different from the operations that compute the memory addresses for loads and stores. Both types of operations have different characteristics, like e.g. dynamic range and different dependence chains. By separating them, their execution can be made more efficient.

As various data optimizations like DTSE [Pan01, Cat98b] substantially increase the addressing complexity, the address calculations can consume a significant amount of resources and should be looked at in detail. Hence, platform-independent source code transformations have been proposed to reduce this overhead [Mir96]. On top of that, a separate platform-dependent compiler phase has been added in the flow of [Rag09b] and [Lam09].

Architecture Most processors perform data and address computation operations on the same slots. As the operations that compute the addresses and the data computations of the application algorithm follow separate dependence chains, they can be separated onto different sets of slots. Only the load/store operation forms a synchronization point between the two paths.

Address calculations have different characteristics than operations on data. The dynamic range of calculations on addresses (fixed range of, e.g. 16 bits depending on memory size) and iterator values is not necessarily the same as the dynamic range of the data (e.g. 8-bit data for pixels). Separating the datapath and address path enables the FUs to be of different widths. Additionally the instruction set of address and datapath can be customized, in order to optimally support the required operations. This will lead to larger performance and energy gains.

Implementation Using the same high level layout directives that have been mentioned above, the data path and address path FUs can be grouped with their respective register files (or VWRs), memory interfaces and loop buffers.

A more detailed description of the split between address path and datapath can be found in [Tan08]. Various commercial DSPs as well as research works like [Kim05, Mir97, Par99, Ver98] also exploit such a split between address generation unit and the processing datapath for low power and/or high performance as well.

3.7 Putting it together: FEENECS template

Figure 3.14 combines the optimizations for different components that have been presented in the previous Sections. The presented processor design is still a template and architecture exploration within this template is required to find the optimal architecture for one application or a set of applications. Based on the performance requirements and on initial feedback from synthesis for a certain technology node, the parameters in the template for this in-order processor have to be fixed, especially the number of units, the precise pipelining and the forwarding paths.

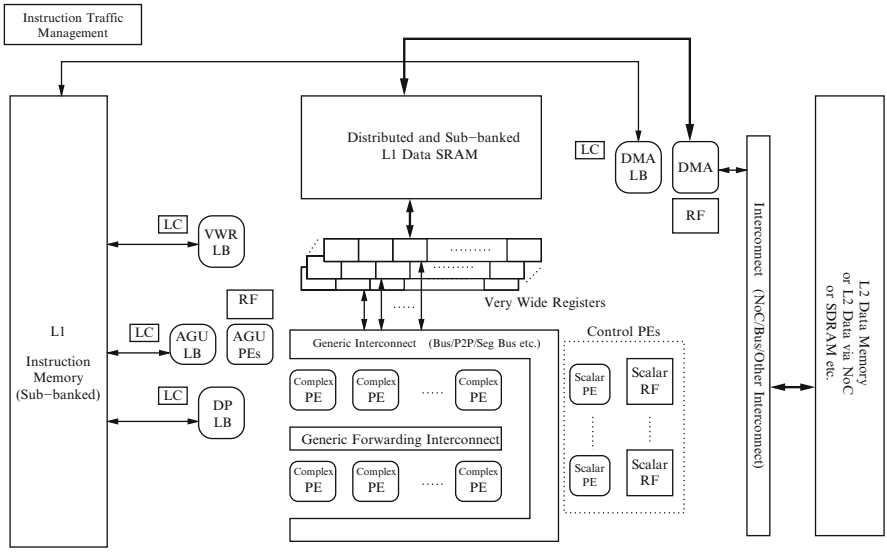


Figure 3.14: Complete high level efficient architecture: FEENECS architecture template

The most notable components of Figure 3.14 are the data memory hierarchy, that consists of an L2 background data memory that is accessed over a generic global communication architecture (NoC, Bus or other). Data is transferred from this L2 memory into the L1 Data memory, a Scratch-Pad memory, by the DMA. From there on, complete lines of the SPM are moved to the VWRs for data parallel computations and single values can be copied to a scalar RF. The AGU units are also placed close to the background or foreground memory units between which they organize the transfers. Preferably they are even split over read and write ports so that they can be fully local to the port they are providing addresses to. The datapath FUs also come in two types: (1) a set of Complex FUs that balance DLP and ILP (a set of SIMD FUs) and use extensive forwarding to reduce VWR accesses, and (2) a set of control FUs that are of scalar type and access the scalar RF. On the instruction side, the L1 Instruction memory can be accessed directly by control code, but for the kernels a set of physically distributed loop buffers are used, placed next to the components they control. To improve the efficiency of the instruction memory, these loop buffers also have distributed control and can synchronize at certain points using local controllers (LCs). As they do not follow a single program counter (PC), the number of NOP (no operation) entries is heavily reduced in applications that exhibit very different activation frequencies for the different components.

The communications between different processors of this type (inter processor communication) is compatible with other related work like [Ler08], while the intra processor connections can be optimized using techniques like segmented bus etc. [Pap06].

In this architecture template, the implementation and layout guidelines that have been discussed in the previous sections have been used to group the different architecture components based on their communication requirements in order to reduce the interconnect cost. Hierarchically structured hardware design using *Datapath Generation* [DPG05] and *Energy-aware floor-planning*[Guo08] can be used to propagate these constraints down to the physical implementation.

This parameterized template of a domain-specific instruction-set processor platform template, offers a potential energy-performance figure of merit of over 900 MOPS/mW in the TSMC 40 nm CMOS technology (using non-optimized standard cell synthesis). Importantly, the basic architectural components of the FEENECS template have a quite wide scope including all domains that are loop-dominated, that exhibit sufficient opportunity for data level parallelism, that comprise signalswith multiple word-lengths (after quantisation exploration) and that require a relatively limited number of variable multiplications. Prime examples of this target domain can be found in the areas of wireless base-band signal processing, multimedia signal

processing or different types of sensor signal processing. See [Lam09, Rag09b, Kri09] for more information. The set of architecture modifications that is presented above has also led to the filing of several patents, as described in [Rag06a, Lam08a, Jay08].

3.8 Comparison to related work

Research groups are mostly focusing on a single component of the platform. Different groups are active in optimizing the on-chip communication architectures (e.g. [Ryu01, Ye02]), the processor power consumption and performance (e.g. [OdB01]) or the cost of the memory hierarchy (e.g. [Ram05] for data and [Pet03] for instruction memory). Often unoptimized, standard implementations are used for the components that are not the main focus, which leads to wrong conclusions on the relative contribution to the platform cost. Little work has been done to consider the complete platform, using optimized implementations for all components. An analysis of overall system performance has been published by [Xu04], but it unfortunately does not consider power.

At the time of study and of publication of our papers [Lam05] and [Lam04] (2004–2005), this was the first effort to compare all parts of a realistic platform in an *apples with apples* comparison. More recently, additional processor styles have been added for a similar application by [Por06]. And also we have performed additional experiments with our COFFEE compiler framework [Rag08b]. These results extend and substantiate the conclusions of this chapter.

3.9 Conclusions and key messages of this chapter

As SoC-based embedded systems grow and become more complex, a comprehensive energy breakdown of such a platform is crucial to evaluate the contribution of the different components, identify the bottlenecks and determine the impact of different power optimization efforts. In this chapter, such a study and analysis is presented for all important parts of a realistic SoC platform. This case study, using a relevant complete application (the MPEG2 decoder) instead of a kernel, provides one consistent story, for which all parts of the platform have been optimized in order to show the real bottlenecks. The power bottlenecks are highlighted and possible techniques to tackle these bottlenecks are discussed.

An overview of the most important embedded processor styles and energy or performance improving techniques is presented. A performance comparison, using the same consistent context of the case study, provides an indication of the relative performance potential the different styles can provide. The conclusions remain valid for the potential hybrid styles that are being used more and more, including heterogeneous processors where the different functional units are widely differing in instruction set choices, or processors augmented with customized accelerators.

Another major focus in this chapter is the design of a scalable high-level architecture design template, targeted at low energy, high performance embedded systems for streaming applications. An important characteristic of the proposed template is the incorporation of the importance of interconnect in future scaled technologies. The problematic scaling of interconnect is taken into account explicitly during the evaluation of cost trade-offs and when proposing implementation (layout) guidelines.

Currently, embedded systems designers are already facing a set of complex trade-offs: energy efficiency vs. performance vs. area. Most often they are working on a single element of the platform and try to optimally design this, considering different design styles to be able to pick the one that is best suited to meet the given requirements. While making this choice, both the impact of the local decision on the other parts, as the effect of the other parts of the system on the local decision has to be taken into account. Solving all these entangled problems together is too complex, but a high level knowledge of the main contributors to the platform cost and the overall trade-offs that are important in every part can help a designer to come closer to the global optimum. In our research we have always tried to aim for globally better solutions when optimizing locally. A similar argument is present for the mapping flow that should accompany such an architecture template. Such a flow is not the focus of this book, but more information about the status of the mapping research is provided in [Lam09], [Rag09b] and the patent filing [Rag08a].

This chapter has shown that embedded platforms consist of many components that each contribute in different ways and different amounts to the total cost. Therefore, it is important that the relations between these components and the effect of local optimizations on other parts are taken into account. The energy breakdown presented in this chapter and the relations that have been identified together with the resulting template, will now be used throughout the following chapters to decide which optimizations make sense in a certain context and which have to be investigate further.

CHAPTER 4

Overall Framework for Exploration

Abstract

This chapter presents one of the core contributions of the book which also forms one of the main stepping stone for the rest of the book. It presents the compilation, simulation and energy estimation framework for modeling the large architecture design space of single core platforms for low power embedded systems. For multi-core platforms, the intra-core space can be reused but the inter-core data memory and communication network organisation should be added. The framework has been made to be consistent and complete with respect to the architecture space commonly used in embedded systems including possible extensions needed for future evolution. The chapter also presents some exploration results which are obtained using the framework. A few counter-intuitive trends which emerge from an exploration in this framework are also included.

4.1 Introduction and motivation

In current design flows, energy estimation in an industrial design is mostly performed only at the final stage of the design, when the hardware is nearly completely fixed and gate level simulations are possible. This approach

F. Catthoor et al., *Ultra-Low Energy Domain-Specific Instruction-Set Processors*,
DOI 10.1007/978-90-481-9528-2_4,

restricts extensive energy driven architecture exploration, taking into account the impact of the compiler and possible transformations on the source code. When optimizing applications for a given architecture, designers try to minimize the energy consumption by improving other metrics like a reduction in the number of data memory accesses. For instance in most cases loop splitting and loop unrolling are used as loop transformation techniques which improve the performance and reduce the number of accesses to the data memories. However, these transformations increase the number of instructions and thus adversely affect the instruction memory efficiency. In another scenario, increasing the granularity of SIMD units (Single Instruction Multiple Data) has the advantage that the number of instructions can be reduced thus improving the instruction memory energy. However, the energy consumption of the data-path units and pipeline registers increase. In order to correctly perform these complex global trade-offs, the energy model has to be *accurate/detailed* enough to show the impact of certain optimizations and *complete* enough so that all the relevant aspects of the processing system are considered. Furthermore to explore the high level architecture space discussed in Section 3.6 such a consistent framework is essential.

Decisions at different levels of abstraction have an impact on the efficiency of the final implementation, from algorithmic level choices to source level transformations to compiler optimizations and all the way down to micro-architectural changes as described in the previous chapter. At each abstraction level different tools exist to perform exploration only within that abstraction level. For instance source to source code transformation tools exist, which enable the designer to perform high-level source transformations. However, the gains due to the transformations are typically estimated on a host processor/platform, which is often not the targeted processor/ platform. Hence, the estimation of the gains due to the transformations will not be consistent with the actual gains on the target processor/platform. This requires an energy model of the target processor/platform. In order to keep the performance and energy estimations *consistent*, the tools at different abstraction levels and inside one single abstraction level, must be integrated into one framework.

Besides the large space across abstraction layers, architecture exploration needs to be performed before arriving at a close to the optimal architecture. Addressing the above needs, this chapter presents an integrated exploration framework called COFFEE as shown in Figure 4.1. This framework provides performance estimates either from profiling the application or through instruction set simulation. It also provides an energy estimate of the target architecture as it has an energy model of each component and by estimating the number of activations for various components. Such a consistent and complete framework allows a good basis for architecture exploration. The underlying compiler and the simulator can support different state of the art and advanced low power architectural features, which makes the framework

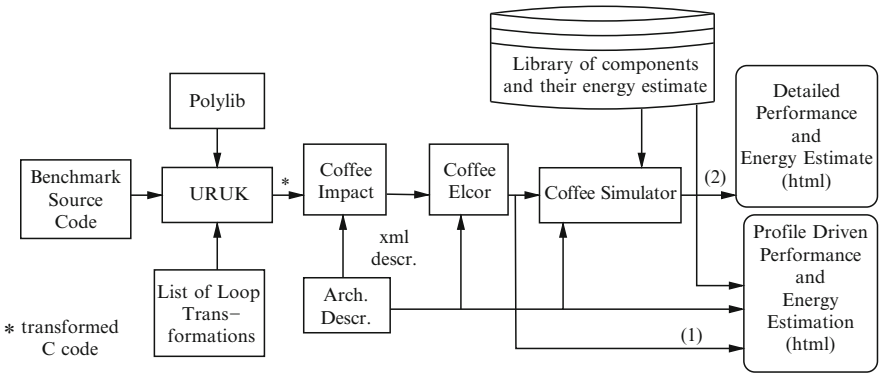


Figure 4.1: COFFEE compiler and simulator framework for transformations, simulation and performance/energy estimation

widely retargetable. New instructions can be added to the instruction set when they are described in an intrinsic library. These intrinsics correspond to the equivalent of an assembly-level mapping decision (or hint) which is incorporated in the high-level C code specification. New functional units can be characterized and added to the library of components to support energy estimation. These features make the proposed framework *extensible*. As it can be seen in the figure, code transformation tools, compiler and the simulator are integrated into one framework. All the relevant information is passed from one phase to the other and the performance and energy estimates are obtained on the targeted platform, making the estimates *consistent*. Furthermore, the energy estimation is *complete* since all the relevant components in the processor system are included in the energy model library all from the same source (technology vendor). Very often estimates obtained from various sources (technology vendors, technology processes, IP vendors etc.) are combined which requires extrapolations etc. which may lead to inconsistencies in the results. Furthermore, the energy per activation figures are obtained through conventional hardware design flow making the energy estimates quite accurate.

The prominent messages for this chapter of the book include:

1. A consistent framework (at ISA level and profiling-level based estimates), which is automated to a large extent, to quickly explore different design options in code transformations, compilation and architecture spaces and also obtain performance, energy and area estimation.

2. A complete energy estimation framework that takes into account all the relevant parts of a processor system namely: data memory hierarchy

and instruction memory organization, functional units (custom and general purpose), register files (foreground memory organization and pipeline registers.

3. A framework that enables simulation and compilation for a wide range of state-of-the-art processors and advanced low power architectural extensions like (clustered) loop buffers, stream register files and distributed data memory hierarchies.

4. And finally, using such a framework this chapter also provides various relevant insights from the different architecture explorations.

The rest of this chapter is organized as follows: Section 4.2 introduces the proposed framework, describes the modeled components and the exploration space that can be covered by the framework. Section 4.3 illustrates the how the energy model for the different components of the platform has been estimated. Section 4.4 compares the proposed framework with existing frameworks for architecture exploration and highlights the similarities and the differences. Section 4.5 describes the architecture space exploration for various benchmarks and provides a performance-energy-area trade-off for various large applications. This section also elaborates the different trends from the architecture exploration and also discusses the time required for the exploration. Finally, Section 4.6 concludes the chapter and summarizes the main messages of this contribution.

4.2 Compiler and simulator flow

Figure 4.1 shows the retargetable compiler and simulator framework. For a given application and a machine configuration, the flow is automated to a large extent, requiring minimal designer intervention. Manual steps are only needed for inserting specific *intrinsics* from the intrinsic library (see above) or in the case of specifying a particular loop transformation. Since the framework is retargetable, it facilitates exploring different machine configurations for a given application.

The loop transformation engine is part of the Wrap-IT/Uruk framework [Coh05], which is integrated into the tool chain (shown in Figure 4.1) and forms the first part of the proposed flow. Essentially, this engine creates a polyhedral model of the application, which enables automating the loop transformations. The compiler and the simulation framework are built on top of Trimaran [Tri99], but are heavily extended in order to support a wider range of target architectures and to perform energy estimation. The integrated and extended flow forms the COFFEE framework.

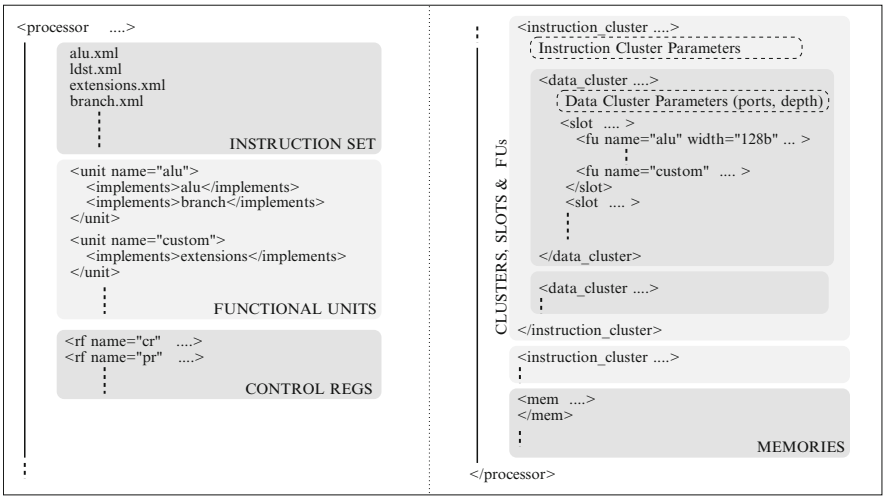


Figure 4.2: XML machine description: Various details like the ISA, functional units, slots, instruction memory hierarchy, register files (data clusters), data memory hierarchy are all defined in the xml file

The application code is presented to the flow as ANSI-C code. A user-friendly XML schema is used to describe the target architecture (machine description). It is read in by processor aware parts of the flow, e.g. the compiler (Coffee-Impact, Coffee-Elcor), simulator (Coffee-Simulator) and the area/ power estimation engine. The application code can be transformed by the Uruk front-end, before it is passed on to the rest of the compiler and eventually used for simulation. The simulator generates detailed trace files to track the activation of the components of the processor. These traces finally are processed by the area, power and performance estimation engine. Figure 4.2 shows a high level overview of the XML schema used for describing the processor, further described in the following subsections.

The exploration space of the compiler and the processor architecture simulator that is currently supported by COFFEE framework is described below. Note however that this framework is also extensible in terms of adding new architecture components to the energy model library, architecture simulator and the intrinsics library (see above).

4.2.1 Memory architecture subsystem

The simulator supports both Harvard and Von-Neumann based memory architectures. The supported features for the data and instruction memory are described below.

4.2.1.1 Data memory hierarchy

The extended simulator supports complete data memory hierarchy simulations, both for software controlled caches (scratchpad memories or SPMs) and hardware caches. The presented tool-flow assists the designer when selecting a cache or a scratchpad, by reporting energy and performance results. In this way application characteristics can be taken into consideration during this choice.

Scratchpad memories have been shown to be extremely efficient in terms of energy, performance and area for relatively static applications [Ban02]. For scratchpads, the data transfers to and from the higher level memories are handled by the DMA. When using scratchpads, the DMA has to be programmed to perform the correct data transfers and the correct timing is accounted for. An intrinsic library has been added to provide an interface to the designer to program the transfers to and from the higher level data memory, e.g. *DMA_TRANSFER (source_addr, spm_dest_addr, transfer_size)*.

This is similar to how state-of-the-art scratchpads and DMAs are controlled. Typically the insertion of these intrinsics is performed manually by the designer. However, in this framework the insertion of DMA intrinsics has been automated when the loop transformation engine is used (further explained in Section 4.2.3). The DMA can also support more complex functionality, like changing the data layout during the transfer, interleaving or tiling data. This type of advanced DMA can help to reduce the power consumption further. For caches, hardware cache controllers manage these memory transfers. The choice for either cache or scratchpad depends on the dynamic or static nature of the application, and should be made by the designer based on simulation analysis [Abs07]. In both cases, the energy and performance is appropriately accounted for by the simulator.

Another important design decision to be explored is the size and the number of ports of memories and the usage of multi-level memory hierarchies. These decisions heavily affect the overall power consumption of the system and hence it is crucial to be able to explore the entire design space. The COFFEE simulator has been extended from the original Trimaran 2.0, in order to accurately simulate multi-level cache and scratchpad based hierarchies. Connectivity, sizes, number of ports etc. are specified in the XML machine description and can be easily modified for exploration. In case of a cache, the line size, associativity, the replacement policy are also mentioned in the tags of the mem section. Based on the replacement policy (Least Recently Used or Least Frequently Used etc.) the simulator, simulates appropriately. The details of this would be entered in the memories section (e.g. as `<mem>`) of the XML description as shown in Figure 4.2.

Identification of copy candidates for transfer between memories could also be aided by IMEC's MH tool. Also other DTSE techniques can be reused from earlier research [Bro00a, Cat98b] activities as they are source to source transformation tools.

4.2.1.2 Instruction/Configuration Memory Organization/ Hierarchy (ICMO)

An instruction memory is activated every cycle to fetch new instructions. Especially in wide architectures, like VLIWs, this can be one of the most energy consuming parts of the system. Design space exploration of the ICMO can therefore have a large overall effect on the processor energy efficiency. The ICMO can be simulated as a cache, a scratchpad or a multi-level hierarchy. Other advanced features, like Loop buffer (or L0) clustering and loop counters (e.g. [Gor02a, Jay05b]), are also supported. Loop buffers can also be clustered to reduce the power consumption [Jay05a, Kob07b]. The precise L0 clustering would be under the `<instruction_cluster>` tag of the XML, as in Figure 4.2 and the parameters like width and depth of the loop buffers are described in *L0 Cluster Parameters* section as shown in Figure 4.2.

Figure 4.3 shows different supported configurations of the instruction memory. (a) is a conventional L1 configuration where the Program Counter (PC) fetches instructions from the L1 instruction cache and executes them on the FUs. (b) shows a centralized loop buffer, where the loops are loaded from the L1 instruction memory to the loop buffer when the loop starts. During the loop execution, the LC (Loop Controller) fetches the instructions from the loop buffer instead of the L1 memory. (c) shows a distributed loop buffer organization that can be customized to the application loop size for every slot to minimize energy consumption, but are still controlled by a single LC. The COFFEE framework supports automatic identification and loading of loops into the loop buffers. Compilation and design space exploration for distributeds loop buffers is described in detail in [Jay05b, Vda04a]. More

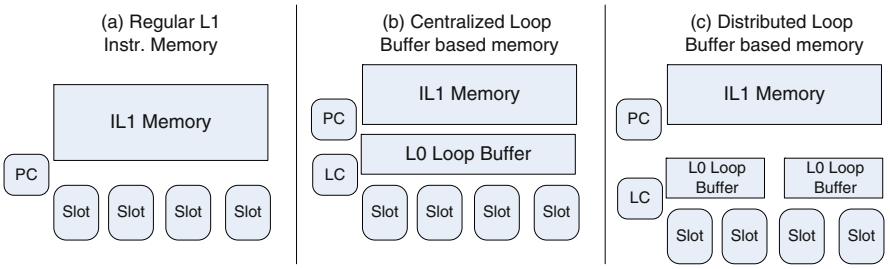


Figure 4.3: Variants of Instruction Memory Configuration are supported in COFFEE

complex loop buffer organizations, where every loop buffer is controlled by a separate LC (described in Chapter 6), are also supported. For all these cases, compilation, simulation and energy estimation are supported.

In summary, we propose the remove the energy bottleneck in the instruction memory organisation by partitioning it into smaller and heavily distributed units that are heavily accessed but that require much less energy per access. The overall control still resides in the large L1 layer organisation but that is nearly not accessed any more. Moreover, the smaller units are located very close to the platform components that consume their control bits. Hence the wiring contribution is significantly decreased. Finally, also the internal overhead in these local units is heavily reduced by separating the distinct functionality (e.g. addressing versus arithmetic operations) over different loop buffers that can support incompatible loop nests with zero overhead (local) condition and loop execution. The down-side is the increased compiler complexity for mapping a given code onto this organisation, which requires also non-traditional techniques that have not been available in conventional compiler literature. But also on that side of the research, significant progress has been made (see e.g. [Kob07, Kob07b, Tan08, Tan09]). This is not the topic of this book though.

4.2.2 Processor core subsystem

The processor core subsystem consists of the datapath units and foreground memory organization/register file. These components are described below.

4.2.2.1 Processor datapath

The proposed framework supports a large datapath design space. Different styles of embedded processors can be modeled, from small RISC processors with a single execute slot,[1] to wide VLIW processors with many heterogeneous execution slots. Multiple slots can execute instructions in parallel, and can internally consist of multiple functional units that execute mutually exclusively. The number of slots and the instructions that can be executed by each slot (this depends on the functional units in that particular slot) can be specified in the XML machine description (as shown in the `<slot>` section in Figure 4.2). New functional units can be added to the architecture, compiler and simulator easily by modifying the machine description and adding the behavior of the new instruction (in C) to the *intrinsic* library. The operation's latency, operand specification, pipelining, association of the functional unit to a certain slot, are specified in the XML machine description and correctly

[1] Note that in this book the term "slot" always refers to an execute slot.

taken into account during simulation. Different datapath widths can be supported: 16-bit, 32-bit, 64-bit, 128-bit or even higher. By varying the width and number of slots, the trade-off between ILP and DLP can be explored. The width can be specified for each FU separately, allowing the usage of SIMD and scalar units in one architecture. An example of this approach is shown for ARM's Cortex A8 [Bar05b] and for other novel SIMD architectures with forwarding networks [Sch07] in the appendices of [Rag09b].

The pipeline depth of the processor can be specified in the machine description and the compiler correctly schedules operations onto pipelined functional units. It is assumed that the processor consists of a standard pipeline with Fetch, Decode, multi-cycle deep Execute and Writeback stages. Based on the activity, the energy consumption of the pipeline registers is automatically estimated. This is crucial for architectures with deep pipelines (high clock frequency), and for wide SIMD architectures. In both cases the number of flip-flops in pipeline registers is large and accounts for a large amount of the energy cost.

4.2.2.2 Register File/Foreground Memory Organization

Register Files or Foreground Memory are known to be one of the most power consuming parts of the processor. Hence it is important to ensure that the foreground memory design space is explored properly. The COFFEE flow can handle several foreground memory options including centralized register files, clustered register files, with or without a bypass network between the functional units. The size, number of ports and connectivity of the register files/foreground memory are specified in the machine description file as shown in the *Data Cluster Parameter* section in Figure 4.2.

Forward paths exist between the functional units and in some architectures like the SyncPro [Sch07] architecture, to transfer data from one cluster to another after the Execute pipeline stage. For such paths, the compiler and simulator can be re-targeted. The modeling and exploration of a few such architectures is detailed in the appendices of [Rag09b].

Figure 4.4 shows different register file/foreground memory configurations that are supported by the framework. Combinations of the configurations shown in Figure 4.4 are supported, both in the simulator and the register allocation phase of the compiler. Separate foreground memory for scalar and vector slots can be used together with heterogeneous datapath widths. An example of such a scalar and vector register file/foreground memory is shown in appendices of [Rag09b] using ARM's Cortex-A8 core [Bar05b]. Various mechanisms for communication of data between clusters are supported, as shown in Figure 4.4, ranging from extra copy units to a number of point to point connections between the clusters. The performance vs. power trade-off

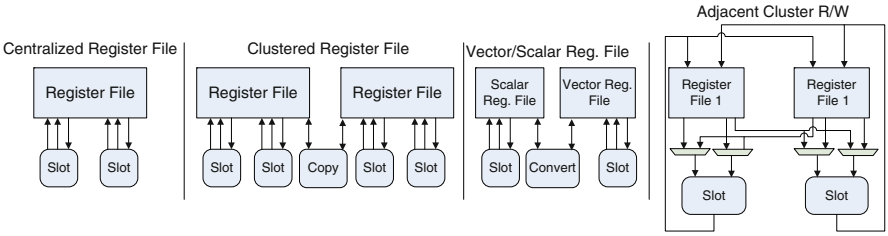


Figure 4.4: Variants of register file architectures that are supported in COFFEE

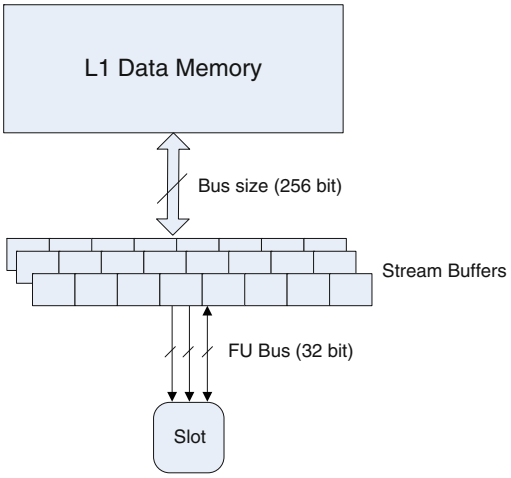


Figure 4.5: Stream register file architecture

(as inter cluster copy operations take an extra cycle, while increasing the fan-out of register file ports costs energy in interconnections) can be explored using the framework.

In case the processor has multiple Functional Units (FUs) that can run in parallel and multiple distributed L1 data memories, there can be a need to go beyond a point-to-point or shared bus towards a more segmented bus like solution [Guo08, Pap06]. The framework however does not support segmented bus based approach yet. For the rest of the book, either point-to-point or a shared bus based solution is used.

Another commonly used foreground memory/register file architecture is a stream register/buffer. Such architectures have been used in [Dal04, Rag07a]. The precise micro-architecture of such a stream register is one of the core contributions of this book and is further detailed in Chapter 8. One such example is also shown in Figure 4.5. The stream register architecture consists of two parts: the data memory (known as SRF/Stream Register

in [Dal04]) and the stream buffer (known as stream buffer or Very Wide Register/VWR (in Chapter 8). Such architectures for the register file are extremely efficient (power and performance) for streaming applications. They may have one or more ports towards the datapath. It is important that such efficient architectures are also covered by a design space exploration framework and in COFFEE that is the case.

4.2.3 Platform dependent loop transformations

Before starting the mapping process on the target platform architecture, it is essential to perform a number of enabling transformations. In literature several classes of such compiler oriented transformations have been described. However, in addition to the ones that are directly coupled to the compiler or low-level mapping process, it has become clear that also platform independent transformations are crucial prior to the platform-dependent stages. A prime example of this are the source code transformations that are required to optimize the usage of the data memory organisation (see, e.g. the DTSE methodology in [Cat02]). These are far from trivial to apply manually because they are quite error-prone to execute in a complex code, and their exploration space is too huge to effectively overlook by a human designer. Hence, we will now explore the frameworks that are required to better support that stage.

Having an automated transformation framework integrated to the back-end compiler is crucial for efficient optimization. The URUK framework [Coh05] performs designer directed source level transformations to optimize locality in the data memory, improve the number of Instructions Per Cycle (IPC) and to enable vectorization. Automated transformations provided by the framework include: loop split, loop fusion, loop interchange, loop tiling etc. These transformations may be platform dependent or independent. The platform independent transformations however can be performed earlier in the design flow and also can be steered by higher level metrics [Kje01, Hu07]. The designer needs to specify the required transformation in a predefined format and the URUK tool performs the transformations automatically, which is less error prone. We have extended the URUK framework, such that it can automatically insert the DMA transfers at the appropriate points in case a scratchpad is used. A case study of the transformation is performed in the appendices of [Rag09b], where URUK has been used for transformations which optimize locality and enable vectorization. This allows the designer to get early feedback on the effect on energy consumption and performance of code transformations for state-of-the-art and experimental architectures. The designer can judiciously iterate over different (combinations of) transformations, like proposed in [Gir06], to optimize the energy and/or area and/or performance for the final platform.

4.3 Energy estimation flow (power model)

The architecture exploration that we envision for our embedded and largely mobile target application domain, should be strongly energy-aware while at the same time meeting the performance requirements. Traditionally however, such an exploration is targeted to optimizing the performance and decreasing the cycle count, where the power is then reduced in a second stage. We want to turn this around and take care that very performance oriented but power hungry options are not withheld during the exploration. Hence we need a sufficiently accurate but fast energy estimation technique/model to support this goal.

In order to get fast and fairly accurate energy estimates, to guide code transformations or architecture exploration, we propose an energy estimation engine coupled to the instruction set simulator. At this abstraction level, the full hardware description is not yet needed and therefore exploration can be fast. To enable early estimates with sufficient accuracy for relative comparison, we propose the following methodology.

The components of the processor (register file, ALU, pipeline registers, Multipliers, Instruction Decoders, etc.) are up-front designed at RTL level (optimized VHDL description). This is done for various instances of each component, e.g. various register file configurations, in terms of ports, number of words and width.

Once the VHDL for a component with a particular parameter set is available, the description is used as input to the flow shown in Figure 4.6. The target clock frequency of the system is imposed as a fixed timing constraint on the design. UMC90 nm general purpose standard cell library from Faraday [Far07] has been used for all experiments shown in this chapter. However this is not a restriction and in various sections of this book different technologies

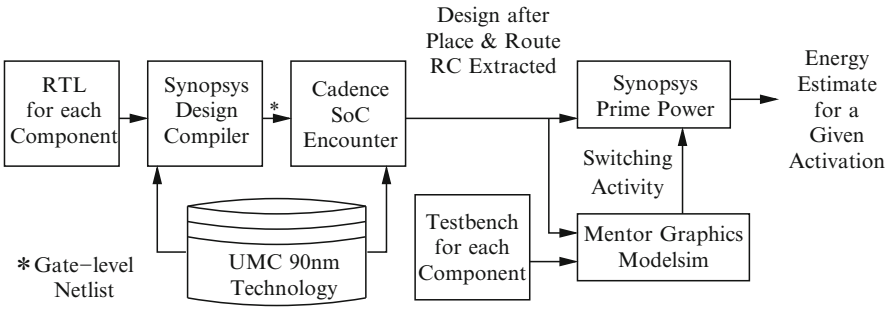


Figure 4.6: Flow used for power estimation for different components of the processor

have been used and have been indicated appropriately. Each component is passed through logic synthesis (using Synopsys Design Compiler [Syn06a]), Vdd/Ground Routing, Place and Route, Clock Tree Synthesis (if needed), DRC, LVS checks (using Cadence SoC Encounter [Cad06]). The extracted netlist (with parasitics) is back-annotated to the gate level netlist in Prime Power [Syn06b]. A testbench is generated for (each instance of) all the components, using input data with realistic toggle behavior. The energy per activation and leakage power for the different components are estimated from the activity information from gate level simulation and the parasitic information. This results in a library of parameterized energy models.

Because memories are highly optimized custom hardware blocks, the standard cell flow cannot be used. Therefore a library of energy consumptions (dynamic and leakage) has been created using a commercial memory compiler (from Artisan [ARM]). Finally, our pre-computed library contains the energy consumption (dynamic and leakage) for various components of the processor using the standard cell flow, and for memories, using a commercial memory compiler. However this can be replaced with more semi-custom design methodologies. Such semi-custom efforts are common for energy critical designs like memories, register files and critical datapath. Efforts for generating these designs in a parameterized way are being carried out at ESAT, K.U. Leuven [Cos07, Cos08] and RWTH Aachen [DPG05, Gem02].

The access statistics needed for energy estimation can be done at two possible levels: after profiling or after instruction set simulation (ISS). The choice between either a profiling based energy estimation or after ISS depends on the accuracy and the design-time needs. For early exploration and pruning, the designer may choose a profiling based energy estimation. Various effects likes cache behavior etc. would not be accurate in the profiling based method. After an early pruning, a more detailed ISS based energy estimation may be done. In case the designer requires a high quality and has sufficient compute power and time available, he/she may choose to run an ISS based estimation to get a more accurate numbers. A more detailed description of the energy model and the different components of energy consumption in sub-micron technology is illustrated in the appendices of [Rag09b]. The approximations and the possible errors introduced in the estimation are also elaborated there.

After compilation, the code is profiled on the host for the given input stimuli. Each of the individual basic/super/hyperblock is annotated with a weight. Weight is the number of times that block would be executed if the same input were to be fed during simulation. Based on the weight of each block, the compiled code and the energy/access of the individual components, the COFFEE tool is capable of estimating the energy consumption of the processor. This is the flow marked as (1) in Figure 4.1. A profiling based estimation is extremely fast as instruction set simulation is not performed and only a

high-level energy modeling abstraction is incorporated. In this case accuracy is traded off for estimation speed with respect to an instruction set simulation flow (described below), because the profiling based estimation is not capable of keeping track of the dynamic effects, like e.g. cache behavior. Profiling based estimation can be used for quick and early exploration.

The instruction set simulation based estimation flow, described in Section 4.2, counts the number of activations for each of the components. Based on this activation and the components' energy/access from the pre-computed library described above, the energy consumption of the complete system is computed (marked as (2) in Figure 4.1). This approach correctly keeps track of dynamic effects and is slower than the profiling based approach, but it is still orders of magnitude faster than a detailed gate level simulation for the complete processor, and therefore fast exploration is possible. Given such a fast exploration a wide range of architecture exploration can be performed sufficiently quickly (few hours).

The power model itself in the tools also supports coarse grain power gating and clock gating and can be used when to duty cycle when the schedules have many holes (no operations to perform). The overhead of this should also be modeled appropriately. The above modeling of the total energy consumption and average power consumption of the complete system would allow the designer to either analyze and optimize the processor architecture (either by a change in architecture or a compiler optimization or transformation).

4.4 Comparison to related work

Various frameworks are available which aim to address the problem of design space exploration for processing sub-systems of embedded systems. Most of these frameworks support some of the desired features but not all important state-of-the-art extensions in all parts of the processor. All the features mentioned in the previous section are essential for a framework to be used for future embedded system designs.

Trimaran [Tri99] is a fast and extensible VLIW compiler and processor exploration framework. The proposed framework is based on Trimaran 2.0, but we have extended the compiler optimization and processor architecture design space to a large extent. Few examples are clustered register file support, stream register file support, clustered loop buffer support, cache simulation and scratch pad memory simulation (a complete list of the design space parameters is described in Section 4.2). In addition we have also extended Trimaran with a sufficiently accurate energy estimation engine (after profiling and after instruction set simulation) and also coupled a loop transformation engine in a consistent manner.

ACE's CoSy framework [ACE08] and GCC [GCC07] are retargetable compiler frameworks which are fast and extensible. They also support a wide range of high-level compiler optimizations and also support code generation for a wide range of processors. However, these frameworks do not support instruction set simulation and energy aware exploration. GCC is mainly targeting code generation for general purpose oriented processors (like x86, Alpha, PowerPC, etc.) rather than for low power embedded processors, which is our focus. The road map of GCC extensions indicates this is slowly changing, e.g. providing support for loop transformations and vectorization. One of the possible future directions of the framework is integrating the energy aware exploration framework with GCC as a front end. Other frameworks, like CoSy [ACE08] target low power embedded processors, however their scope of retargetability is limited.

In the code transformation space SUIF [Sui01] pioneered enabling loop transformations and analysis. Wrap-IT [Coh05] from INRIA uses the polyhedral model for analysis and to perform transformations. These tools alone are not sufficient as transformations can have an impact (positive, negative or neutral) on various parts of the processors. This is because these transformations are not platform independent and therefore essential to have one integrated flow. In the proposed flow, the Wrap-IT loop transformation framework has been directly coupled to the retargetable compiler, simulator and energy estimation engine.

In a sense, the above frameworks are complementary to the proposed COFFEE framework since all three have the capability to perform performance explorations in a certain sub-set of the intended design space consistently. Essentially compared to these, the COFFEE framework supports a larger design space along with an accurate, complete and consistent (w.r.t processing system) energy estimation engine.

Wattch [Bro00b] and SimplePower (which is based on Simplescalar [Aus02]), enable architectural exploration along with energy estimation. However, their power models are not geared towards newer technologies, their parameter range is still too restricted and they are geared towards high performance systems. To the best of my knowledge, these frameworks do not support important architectural features like software controlled memories, data parallelism or SIMD (Single Instruction Multiple Data), clustered register files, stream register files, loop buffers etc. These features have become extremely important for embedded handheld devices as energy efficiency is a crucial design criterion.

Epic Explorer [Asc03] is VLIW exploration framework, also based on Trimaran [Tri99], supports energy estimation of the processing system. However, the energy model is not detailed enough to give a breakdown of energy consumption of the various processor components (e.g. for register

files, functional units and pipeline registers). The architecture design space is also limited to general purpose VLIWs and the memory hierarchies supported are only caches. Furthermore, a loop transformation engine is not coupled to it in order to make a consistent framework in order to explore the design space an embedded designer might wish to cover. Other simulation environments like Liberty [Lib02] also exist which allow exploration of various micro-architectural features. However Liberty is a simulator environment and also lack a compiler and an energy model. However the simulation part of the proposed framework could be built using Liberty.

Other industrial tools like Target's IP Designer [Tar08], Coware's Processor Designer [CoW08a] and Tensilica's XPRES [Gon02], provide architectural and compiler retargetability, but the supported design space is limited to a restricted template and they do not provide fast, high-level energy estimates. Synfora's PICO Express does provide early estimates for architecture exploration, however, it does not support various architecture features like streaming registers, loop buffers etc. Detailed energy estimates can be obtained by synthesizing the generated RTL and using the traditional hardware design flow. Generating such detailed estimates is too time consuming for wide exploration and for evaluating compiler and architectural optimizations. Energy estimates based on a library of architectural components, as proposed in this book, do not suffer from these drawbacks, as they are fast and sufficiently accurate and complete for the early explorations of an embedded designer.

Several studies in literature describe the outcome of a the manual exploration of a specific subdomain in the entire exploration space. For instance, a detailed study of foreground memory organization and the impact on power and performance has been performed in both [Rix00a] and [Gan07]. But these studies are limited to the specific domain (e.g. foreground memory/register file alone) and do not provide a framework for exploring other parts of the processor.

In the analytical power estimation domain there exist various research works like [Ben02] which is limited to the Lx processor, [Sch04] which is limited to the TI C6x processor, [Tiw94] which is limited to i486 and Sparc, [Cha00, Sin01] which are accurate but limited to ARM only, [Ye00] which is limited in architecture space, [Pon02] which is very accurate but uses a old technology and models only the datapath. To the best of my knowledge no framework exists which is complete, consistent, flexible and accurate in its power modeling over a large architectural space with a large amount of state-of-the-art features.

The proposed COFFEE framework combines the benefits of all the above exploration frameworks, while giving fast and accurate estimates of energy and performance early in the design process. It also provides a framework which

can analyze the impact of high level code transformations on the architecture's performance and power consumption. This framework can be used to explore hw/sw co-design, architectural optimizations or software optimizations. Note though that *it is not intended that* COFFEE *would produce the final assembly code* on an instantiated platform. The later is the focus of commercial tools like Synfora's PicoExpress [Syn08], Target's IP Designer [Tar08], Coware's Processor Designer [CoW08a] and other such tools. Once again this could introduce some amount of absolute inaccuracy in the estimation due to the way VHDL is written or other such cases.

4.5 Architecture exploration for various algorithms

The previous sections described the large architecture space that can be covered by the COFFEE framework. This section illustrates the exploration and the impact on energy, area and performance. Section 4.5.1 shows an exploration of various parameters: number of data clusters, number of slots per cluster, depth of the loop buffer and size of the data memory. Section 4.5.2 will also illustrate various counter-intuitive trends which emerge whilst covering the complete architecture space. Such trends are often hidden when only one part of the architecture is explored.

4.5.1 Exploration space of key parameters

To cover the large number of parameters that are varied a naming convention has been introduced. Table 4.1 shows the naming convention used in this

Naming Convention	Range explored	Details
dc=2	1,2,3,4	Two data clusters
spdc=4	2,4	Four slots per data cluster
dcd=16	16,32,64	Sixteen registers per data cluster
icd=32	32,64,128	Thirty two entries in Loop Buffer
mem=8K	4–256 K	8 KB data memory (with 2 bank, 4KB/bank)
dc:spdc:dcd:icd:memk		Generic architecture name
2:4:16:32:16K		Two data clusters with four slots/data cluster with 16 deep register file/data cluster with 32 entry deep loop buffer and 16 KB (4×4KB) data memory

Table 4.1: Naming Convention Used for Architecture Exploration

section and the range over which each of these parameters are varied. The different parameters that are varied are:

1. Number of data clusters: This corresponds to the number of clustered register files. The interconnection network between the clusters as shown in Figure 4.4 (clustered register file) is assumed for experiments.
2. Number of slots per data cluster: This corresponds to the number of issues slots used per data cluster. In case the number of slots increases by one, the number of ports required on the corresponding register file increases by two read ports and one write port.
3. Register file size: This corresponds to the number of registers available per data cluster's register file.[2]
4. Loop buffer depth: This corresponds to the number of instructions the loop buffer can store at a given point of time.
5. Data memory size: This corresponds to the size of the L1 data memory. It is assumed that the data memory is banked into banks of 4 KB each.

Besides the above variation in parameters, the following design parameters are kept constant throughout the exploration in this subsection:

1. Five stage pipeline with traditional fetch, decode and three arithmetic operator related stages
2. Latency of multiply operation is three execute cycles (pipelined), ALU is one execute cycle, Load/Store latency is two cycles (pipelined)
3. 32-bit datapath
4. 512 entries of VLIW instructions in the L1 instruction memory (this implies wider the VLIW, larger the L1 instruction memory)
5. All slots are homogeneous
6. All data clusters, which are realized as register files, are of the same size
7. A centralized loop buffer is used as shown in Figure 4.3b

To illustrate the design space, various embedded system benchmarks have been used from Mediabench [MedB], Versabench [Rab04], as well as an in-house optimized video decoder H.264/AVC (Advanced Video Codec)

[2]Note that this chapter does not explore the VWR architecture. The trade-offs for different VWR implementation is addressed in more detail Chapter 8.

[Wie03] and a MIMO (Multiple Input Multiple Output) Loop Compensation have been used. From the Mediabench benchmark suite MPEG2 encoder, decoder and EPIC (image filtering algorithm) have been chosen. From the Versabench benchmark suite 802.11a convolution coder and FM radio have been chosen. An optimized version of H.264/AVC (which is a particular implementation of the MPEG4 decoder) has also been chosen. The AVC decoder is optimized for its CIF frame size (352×240) and 25 fps operation. The MIMO Loop Compensation is part of a multi-antenna Software Defined Radio (SDR) running wireless LAN. Loop Compensation forms part of the baseband processing which decouples the inputs from multiple source antennas (in the benchmark the number of antennas is set to two).

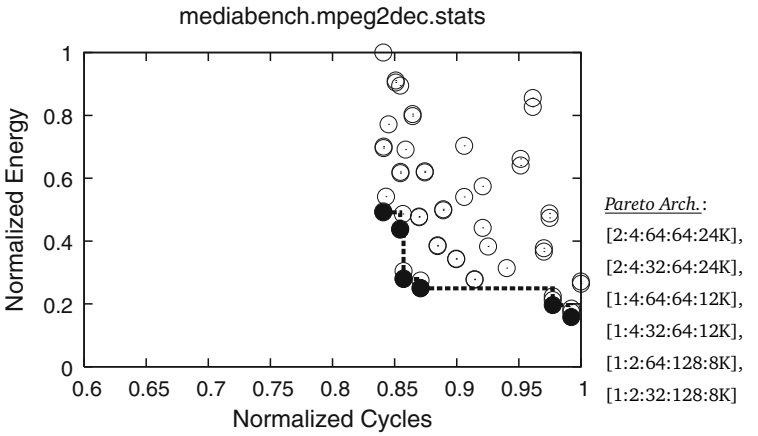
Figures 4.7 through 4.9 show the design space exploration of these different benchmarks. For each benchmark, the figures show the normalized trade-off space between energy consumption and performance. In such a two-dimensional trade-off, points that are further to the right and upward from other points are clearly suboptimal as they add more cost on both axes. So we can discard them. The remaining points are called Pareto-optimal. Hence, only the Pareto-optimal curve for each of these benchmarks is shown in the figures. The Pareto-optimal architectures for each of the benchmarks are then listed in a table next to the figure. The architectures listed on the listing is ordered by from moving from left to right on the Pareto-curve.

It can be seen from Figures 4.7 through 4.9 that a large trade-off variation in energy and performance can be obtained. On the Pareto curves, an average trade off of **55**% in energy and average trade-off of **22%** in performance can be observed. Over all the design points themselves, the energy can be reduced by up to 90% and the performance can be improved by up to 60%. Such a higher level architecture exploration with these trade-offs would allow the designer to prune the design space before the actual instantiation of the architecture.

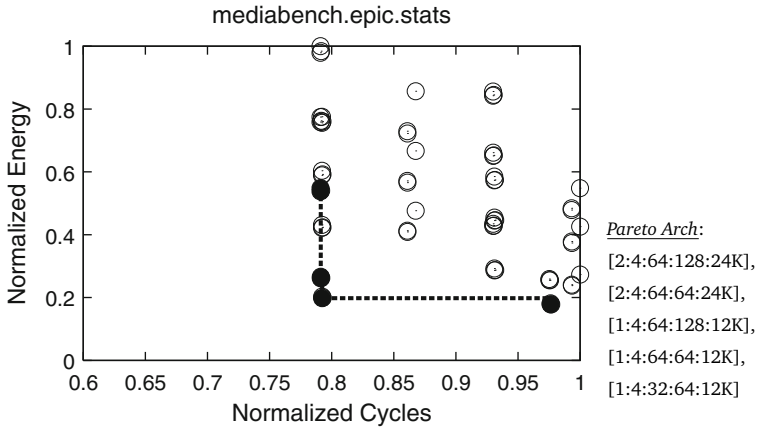
4.5.2 Trends in exploration space

Various trends can also be observed from the architecture exploration. To understand these trends consider the MPEG2 Encode benchmark shown in Figure 4.10. We can observe that architectures with larger number of data clusters[3] and slots like *(1)[4:4:64:128:48K]*, *(3)[4:4:32:128:48K]* provide high performance, while the energy consumption is quite high. The next important aspect is that reducing the register file size reduces the energy consumption, but also degrades the performance. This can be illustrated

[3]Although a cluster can imply either data or instruction cluster, in this section for simplicity we use data cluster and cluster interchangeably. We vary only the data clusters. The number of instruction clusters is set to one.



(a) Mediabench/mpeg2dec Pareto Curves



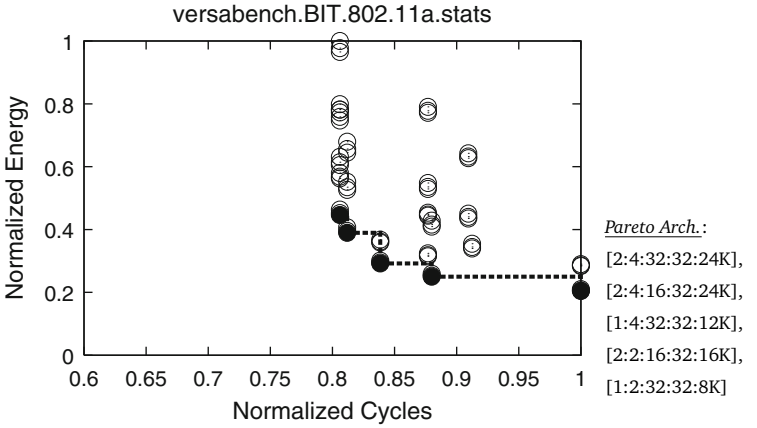
(b) Mediabench/epic Pareto Curves

Figure 4.7: Pareto-curves for benchmarks MPEG2Dec and epic. Note that not all Pareto points are visible and may overlap each other. They are still preserved because they are still Pareto points

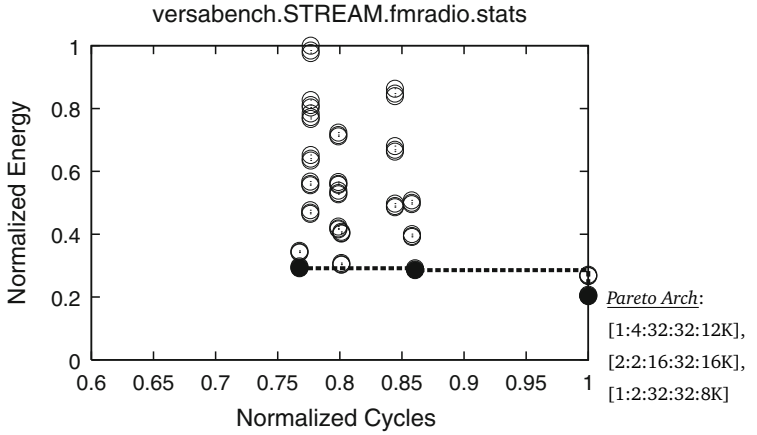
by moving from architecture *(2)[4:4:64:64:48K] to (3)[4:4:32:128:48K]* or moving from architecture *(5)[2:4:64:64:24K] to (6)[2:4:32:128:24K]*. The data memory size also reduces from 48K till 8K as we traverse the Pareto-curve from top to bottom.

More generally one can observe the following trends over the different benchmarks:

1. *Number of data clusters*: For all the benchmarks as one moves from the top to bottom of the Pareto-curve, the number of data clusters



(c) Versabench/bit/802.11a Pareto Curves



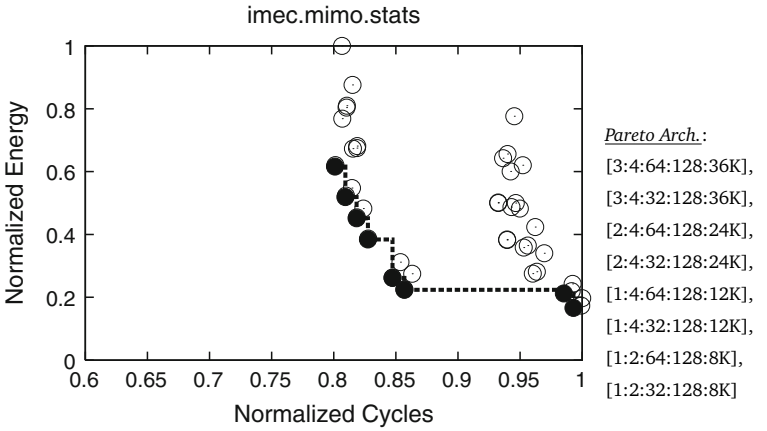
(d) Versabench/stream/fmradio Pareto Curves

Figure 4.8: Pareto-curves for benchmarks 802.11a and fmradio. Note that not all Pareto points are visible and may overlap each other. They are still preserved because they are still Pareto points

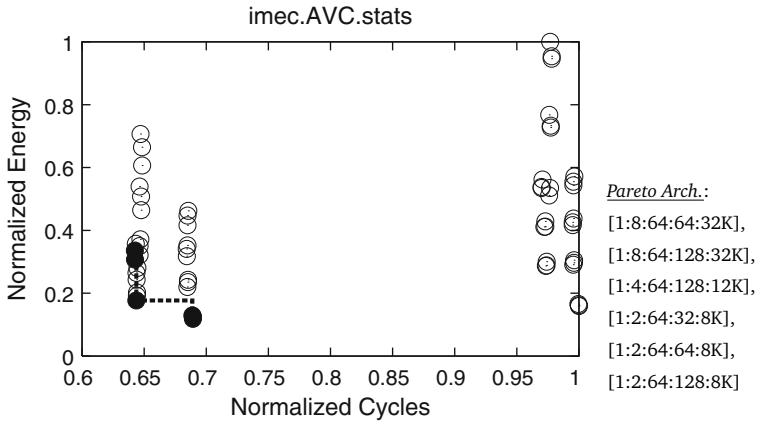
used reduces. This is because as we use lower number of clusters, the performance degrades and the energy consumption also reduces (as the total number of registers and functional units also goes down).

2. *Memory size*: Another common trend that can be observed is that while increasing the number of banks (therefore the total data memory size), the performance improves and the energy consumption also increases.

3. *Register file size*: It can be seen that for most benchmarks, 16 registers/ data cluster is suboptimal. This is because inside the critical parts of



(e) imec/MIMO Loop Compensation Pareto Curves



(f) imec/AVC Pareto Curves

Legend: ——: Pareto Curve
○ : Design Point

Figure 4.9: Pareto-curves for benchmarks MIMO and AVC. Note that not all Pareto points are visible and may overlap each other. They are still preserved because they are indeed Pareto points

the application, the register pressure is high. Furthermore an increase in register file size also improves the performance at the cost of energy consumption. And an increase in register file size causes a substantial increase in energy as a larger number of registers need to be clocked.

4. *Slots/cluster*: Another trend is that as the number of slots/cluster reduces, the performance degrades and the energy consumption reduces.

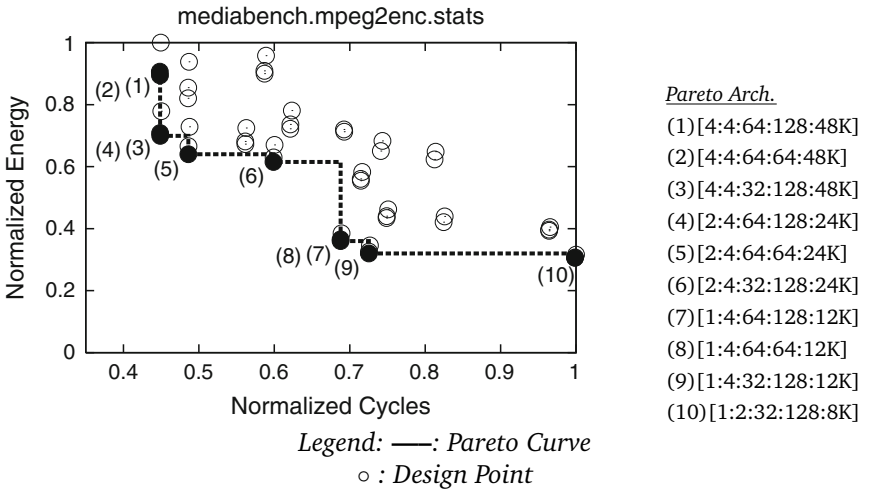


Figure 4.10: Performance-energy trade-off space for MPEG2 encode over the various design points

When the number of slots/cluster is reduced, the number of ports required decreases and therefore the energy consumption of register file also reduces.

5. *Loop buffer depth*: As it can be expected, the size of the optimal loop buffer is benchmark specific. The net size of the loop buffer should be viewed as *Num. of clusters* $\times$ *Num. of slots/cluster* $\times$ *Loop buffer depth*. This value reduces as one traverses the Pareto-curve from top to the bottom.

Besides the above trends, one can observe the following trends which are benchmark specific:

1. *mediabench/epic*: The real trade-off space for epic is limited. This result implies that the compiler mapping is not that efficient. In this particular case the reason was due to an idiosyncrasy in the data cluster assignment phase in the compiler, which performs poorly when many conditional constructs are present within the loops. However these have been solved in the later versions of Elcor which is the backend compiler of Trimaran [Tri08].

2. *mediabench/mpeg2enc* and *mediabench/mpeg2dec*: Both MPEG2 decode and encode have a large variation in the loops sizes, IPC for each kernel etc. This enables different architectures to be better suited for different parts of the algorithm. Therefore the trade-off in the energy-performance space is large.

3. *versabench/stream/fmradio*: This benchmark has several loops which are small with few variables only. This explains the poor trade off present in energy. Moreover this is the only benchmark for which the register file size of 16 entries is still Pareto-optimal. The loop buffer depth required is only 32 entries.

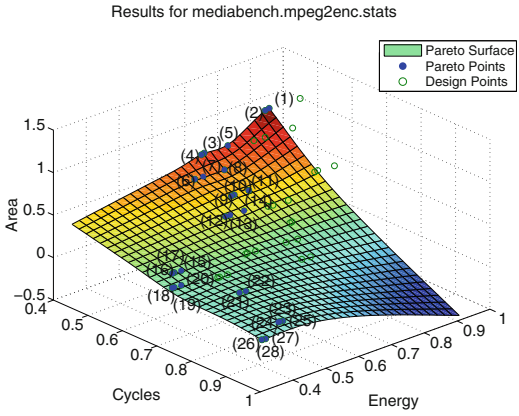
4. *versabench/bit/802.11a*: This initial benchmark is only partly optimized by the designer who coded it. It consists of a loop which in turn contains a function call. But the compiler could inline the function into the loop. Also the loop does not have any loop carried dependency and has a high degree of instruction level parallelism. Therefore a good trade off exists for this benchmark.

5. *imec/mimo*: The loops in the MIMO Loop Compensation algorithm consist of a large number of live variables. This implies that the Pareto-optimal architectures require a larger number of registers. For all the architectures with only 16 registers per data cluster, the compiler could not even find a solution due to the increased register pressure.

6. *imec/AVC*: In AVC the amount of control code is significant. Also inside the loops, a large amount of conditional code exists. Therefore clustered architectures have not been efficiently used and the Pareto-optimal architectures for AVC are all single cluster architectures.

So far the previous sections have presented explorations with two objective functions namely performance and energy. Another important view which is required for system design also includes *area*. Figure 4.11 shows the exploration results on three key objective axes namely: *area, energy* and *cycles* for MPEG2encode. The filled points in "blue" are the Pareto-optimal design points. The "hollow green points" are all the remaining design points. The surface shown is curve fit[4] over the different Pareto-optimal design points. Note that the number of Pareto-points compared to the 2 dimensional plot is higher. This is because of the new objective function (area) which introduces more trade offs between energy and performance. For example, compared to the Performance-Energy trade off space (shown in Figure 4.10), several new extreme architecture points like *(28) [1:2:32:32:8K]* have appeared that exhibit lower area than the other points. A basic trend that can be seen from Figure 4.11 is that, while adding more hardware (area), the performance increases along with the energy consumption. These results validate the basic intuition that area is a first degree estimate of energy. However, some trade off points exist where an increase in area leads to a decrease in energy for the same performance. This kind of result indicates that the compiler mapping performed is not efficient or that the optimizations have reached the point of diminishing returns. By observing this result closely it can be seen

[4]Drawing a three dimensional Pareto-surface is quite non-trivial therefore, a curve fit of three dimensional Pareto-points is presented instead.



Pareto Arch.

(1)[4:4:64:128:48K];(15)[1:4:64:128:12K]
(2)[4:4:64:64:48K];(16)[1:4:64:64:12K]
(3)[3:4:64:128:36K];(17)[1:4:64:32:12K]
(4)[3:4:64:64:36K];(18)[1:4:32:128:12K]
(5)[3:4:64:32:36K];(19)[1:4:32:64:12K]
(6)[3:4:32:128:36K];(20)[1:4:32:32:12K]
(7)[3:4:32:64:36K];(21)[2:2:16:128:16K]
(8)[3:4:32:32:36K];(22)[2:2:16:64:16K]
(9)[2:4:64:128:24K];(23)[1:2:64:128:8K]
(10)[2:4:64:64:24K];(24)[1:2:64:64:8K]
(11)[2:4:64:32:24K];(25)[1:2:64:32:8K]
(12)[2:4:32:128:24K];(26)[1:2:32:128:8K]
(13)[2:4:32:64:24K];(27)[1:2:32:64:8K]
(14)[2:4:32:32:24K];(28)[1:2:32:32:8K]

Figure 4.11: Area-performance-energy design space for MPEG2 encode over the various design points

that moving from architecture *(4)[3:4:64:64:36K]* to *(2)[4:4:64:64:48K]*, the number of clusters is increased, but the performance remains the same and the energy consumption is increased. In this case the application does not have sufficient (exploitable) ILP in order to fill four clusters of four slots per cluster. These results illustrate that the proposed COFFEE framework enables the designer to analyze and observe some interesting and non-trivial trade-offs and also identify the limitations and bottlenecks of the system, optimizations and applications.

Figure 4.12 shows the energy breakdown of the different components for different benchmarks on a subset of their Pareto-optimal architectures as indicated on the x-axis. As in the previous energy breakdown plots, the clock energy of a corresponding component is not shown separately but included in the energy of that component. In Figure 4.12 one can observe a few interesting trends.

For example moving from the architecture *[1:4:32:64:12K]* to *[1:2:64: 128:8K]* for *mediabench/mpeg2dec* causes an increase in the net register file energy as the register file size increases. In contrast, the instruction memory cost decreases as the size of the instruction loop buffer is increased (thereby allowing more loops of the application to be mapped onto the low energy loop buffer).

Consider another example: Moving from architecture *[3:4:32:128:36K]* to *[2:4: 64:128:24K]* for *imec/mimo* causes the instruction memory and

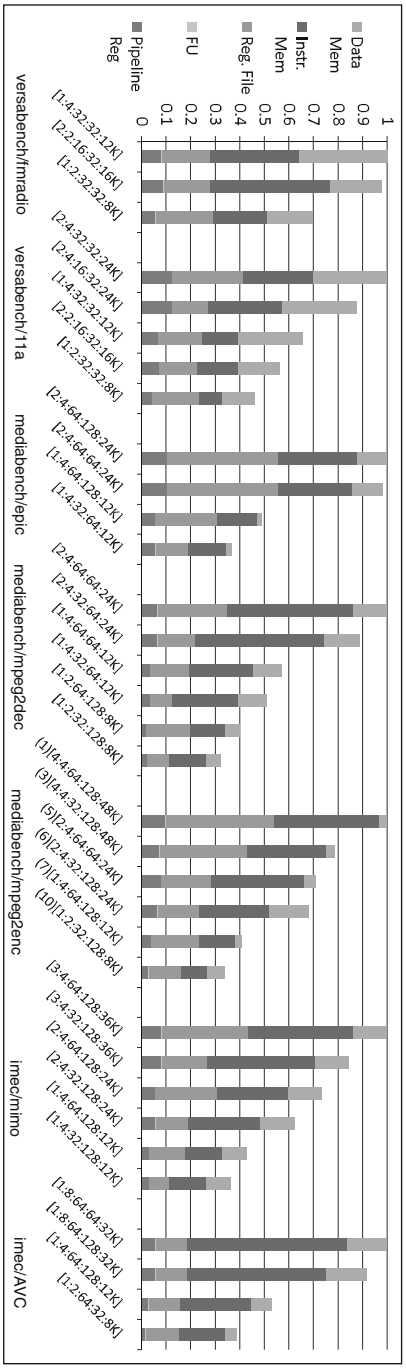


Figure 4.12: Energy split evolution over different Pareto-optimal design points

pipeline energy to decrease due to lower number of issue slots. A larger issue machine requires a wider instruction memory and therefore the energy/access of the instruction memory (despite NOP compression) is higher. However, the register file cost goes up as the number of registers increases from 96 (32 registers/cluster $\times$ 3 clusters) in the first architecture to 128 (64 registers/cluster $\times$ 2 clusters) in the second architecture .

Another interesting trend is that moving from architecture *(5)[2:4:64:64:24K]* to *(6)[2:4:32:128:24K]* for *mediabench/mpeg2enc* causes the instruction memory energy to decrease as loop buffer size is increased. However, the data memory cost increases as the register file size decreases, causing more register pressure, therefore more spilling. This causes more load/stores and therefore the data memory cost increases. This re-emphasizes the claim that it is crucial to have a complete and consistent view of all parts of the system, else various such effects cannot be taken into account in a correct and consistent way.

The above observations are quite non-intuitive to a typical embedded programmer who does not understand enough about the architecture as well as to an architect who does not understand enough about application to comprehend the trends on their own. Also such trends can only be observed using a framework similar to the COFFEE as presented in this chapter, which gives complete, consistent and accurate energy, area and power estimates along with a large set of compiler optimization and architectural options.

4.5.2.1 IPC trends

Instructions Per Cycle or IPC is a commonly used technique for an estimate of performance of an application. Figure 4.13 shows the IPC evolution of the different Pareto optimal architectures for the different benchmarks explored. For each of the benchmarks the bars from left to right represent the different Pareto optimal architectures decreasing in performance (in terms of cycles). A generic trend that can be observed is that as one moves from left to right

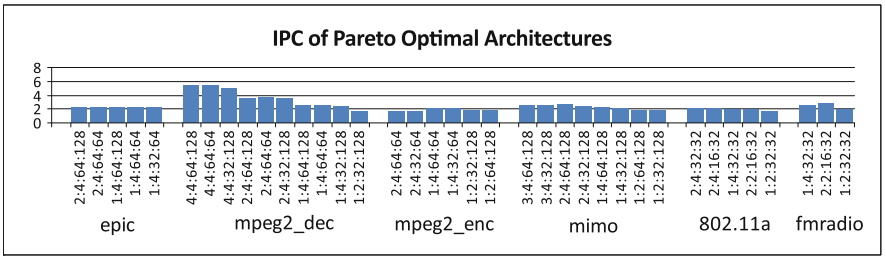


Figure 4.13: IPC analysis of different architectures

the IPC for each benchmark decreases. However, this cannot be generalized and exceptions exist. For example moving from architecture *[1:4:32:32]* to *[2:2:16:32]* for *fmradio* actually increase the IPC, however, reduces the number of cycles. A similar trend is observed when moving from architecture *[1:4:32:32]* to *[2:2:16:32]* for *802.11a* benchmark. All these architectures have four slots, but the second architecture has two data clusters instead of one. Often in case of multiple clusters, data needs to be copied from one cluster to another. This introduces "cluster copy" instructions. While these are valid instructions, they do not add to the performance of the application. Therefore in these cases the IPC increases but the performance decreases.

Another very counter intuitive trend can be seen in case of the *mpeg2enc* benchmark while moving from architecture *[2:4:64:128]* to *[2:4:64:64]*, the IPC increases but the performance decreases. At first sight this would appear incorrect as decreasing the loop buffer size should not affect the IPC. However, as we decrease the loop buffer size from 128 to 64, the number of times the instruction required to activate the loop buffer is higher. For example, since the loop buffer is smaller, there will be many more invocations to activate and load code to the loop buffer compared to the larger loop buffer case. This further emphasizes a need for a consistent framework for exploration.

4.5.2.2 Loop buffers and their impact on Instruction Memory Hierarchy/Organization

Figure 4.14 shows the normalized instruction memory energy consumption annotated on a graph which also contains the ratio of instructions mapped onto loop buffers to L1 instruction memory. For each of the benchmarks, some of the Pareto optimal points have been plotted to observe a few trends. The instruction energy consumption for each benchmark is normalized to the architecture which consumes the highest. For each of the benchmarks as one goes from left to right (as in Figures 4.7–4.9), the performance degrades. This usually means that we go from a processor with larger number of slots to a one with lower number of issue slots. This consequently also means that the width of the instruction memory decreases and therefore the net energy consumption of the instruction memory decreases as well. However, for two processors with the same number of slots, the instruction memory energy consumption also depends on how many instructions can be issued from the loop buffer and how many from the instruction cache. The loop buffer depth has to be optimally sized as well. For example in *epic*, moving from architecture *[2:4:64:128]* to *[2:4:64:64]* reduces the instruction memory energy, but the ratio of loop buffer operations to non-loop buffer operations remains the same. For the benchmark *mpeg2enc*, moving from architecture *[4:4:64:64]* to *[4:4:32:128]* increases the instruction memory energy consumption as the depth of the loop buffer is higher. Therefore the energy/access is also

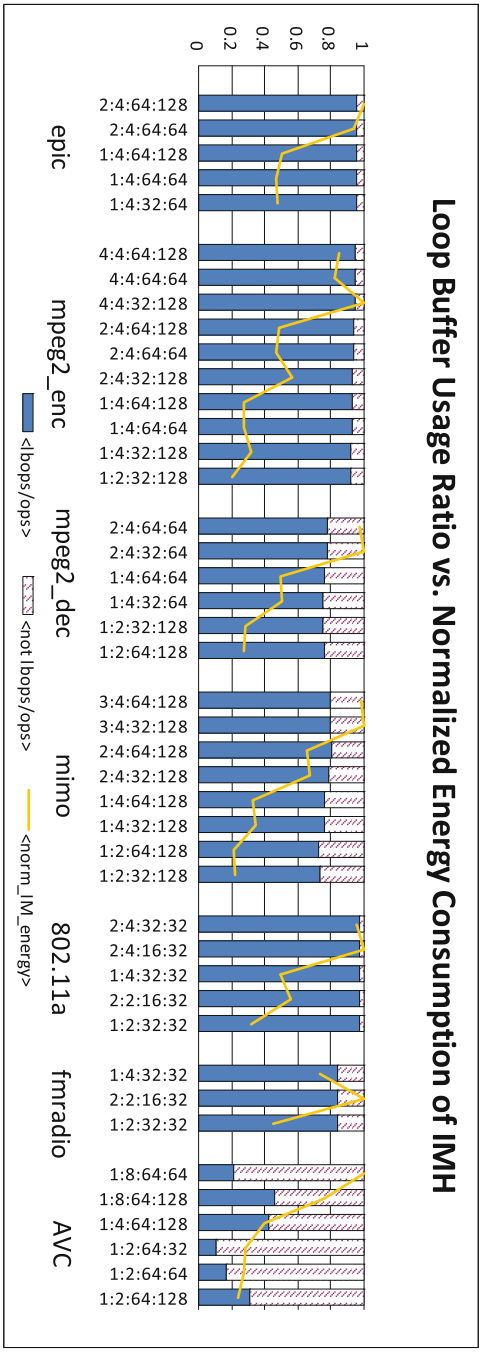


Figure 4.14: Correlation between instruction memory energy consumption and code mapped onto loop buffer

higher. Specifically an odd behavior can also be observed like in the benchmark *fmradio*. The number of issue slots in both architectures *[1:4:32:32]* and *[2:2:16:32]* is four, and the loop buffer depth is 32 entries. However, the instruction memory energy increases. This is because of the clustered register file (architecture *[2:2:16:32]* which has two clusters with two issue slots each) for which not all loops can be software pipelined. In such a case, the compiler uses only the first cluster. Therefore the performance degrades whereas the number of instructions issued increases. This leads to an increased energy consumption.

Such trends further emphasize the fact that it is very crucial to consider the complete processor platform power consumption instead of each component in isolation.

4.5.2.3 Exploration time

To illustrate the complexity of the exploration space, Table 4.2 shows the total time required for performing the architecture exploration, the average time required for one single compilation and the time required to do instruction set simulation on one architecture. The simulation times are reported on a dual core AMD Opteron 2.3 GHz running Red Hat Linux 3 Enterprise. It should be noted that the compilation and simulation time is dependent on the precise architecture chosen and the benchmark. For example the time required on an architecture with few registers and mpeg2dec is quite high. This is because the register allocator tries hard to find a solution. Whereas if an FIR filter was compiled on the same architecture, the time required would be quite low. Another important point to note is that when large benchmarks are compiled on small architectures, the compiler may even fail. In that case it does not lead to a point in the exploration space, but adds to the total time for exploration. However, Table 4.2 shows the average time required

Benchmark	Exploration time (s)	Compilation time (s)	ISS time (s)
versabench/fmradio	1,430	24	97
versabench/11a	1,591	26	1,498
mediabench/epic	5,883	98	412
mediabench/mpeg2dec	9,433	157	3,410
mediabench/mpeg2enc	10,230	208	6,593
imec/mimo	2,234	39	313
imec/AVC	29,579	547	NA

Table 4.2: Exploration, compilation, simulation time consumption

for a single successful compilation run and a single simulation run. The total architecture exploration time is the time required to compile on 60 different architectures to find the Pareto optimal architecture. Once the Pareto-optimal architectures are found after estimation at compilation, instruction set simulation is performed only on the Pareto-optimal architectures.

Special note should be taken of the AVC benchmark. For this benchmark, no single simulation run could be completed because the amount of space required to produce the trace file is larger than 60GB. A disk space utilization of around 60GB (excluding /swap) was reached in a day's time. Unfortunately despite using Linux pipes and optimizing the printing of traces to the bare minimal, a complete trace for AVC could not be recorded and therefore the simulation times are not reported. This shows the need for better and faster simulators.

4.6 Conclusion and key messages of this chapter

This chapter has presented a framework to perform energy-area-performance aware architecture exploration. The proposed framework provides all the necessary low power architecture features to optimize processors for hand-held embedded systems. The accuracy of the energy estimation has been validated by comparing it to a detailed gate level simulation, using an in-house processor design. The chapter also has shown that the proposed framework is capable of compiling, simulating, and estimating energy for a wide range of architectures and advanced low power architectural features. It has illustrated the design space exploration on various benchmarks and industrial applications. Such a framework has shown to provide an energy-performance-area trade-off early in the design phase. Such a framework has been shown to illustrate and explain various counter-intuitive trends that occur during architecture exploration. The COFFEE framework has been used in various parts of this book to perform architecture exploration and to model different architectures. Also it is currently being used by other PhD students and researchers within the current system design activities in IMEC and its network of cooperating institutions.

CHAPTER 5

Clustered L0 (Loop) Buffer Organization and Combination with Data Clusters

Abstract

A distributed L0 buffer organization is an energy-efficient template for augmenting the instruction memory organization at the lowest level of the instruction memory hierarchy. In particular, functional units in the datapath are grouped into logical units called L0 clusters. Each L0 cluster obtains an associated L0 buffer which stores the instructions (coded operations) corresponding to the functional units in that cluster. Also, a local controller in each L0 cluster is responsible for indexing into and progressing through the L0 buffers. In this chapter, that control is however limited to conditions and the loop nest still has to be fully compatible between the L0 buffers. That limitation will be relieved in the next chapter. One of the principal energy-improving aspects in this book is the architecture concept embedded in such a distributed L0 buffer organization. Generating application-specific L0 clusters and compiling for L0 clusters are ways to exploit that distributed organization. In this chapter the repertoire of the organization is described and the key architectural parameters are introduced.

F. Catthoor et al., *Ultra-Low Energy Domain-Specific Instruction-Set Processors*,
DOI 10.1007/978-90-481-9528-2_5,

5.1 Introduction and motivation

Thus far L0 buffer organizations proposed and analyzed in literature are to a large extent centralized, i.e., a single logical cluster is assumed and a single controller controls the indexing into the buffer to store and fetch instructions. However, such an organization in the context of VLIW processors is energy inefficient and its scalability is limited. Firstly, the wordlines of the buffers should be at least as wide as the number of issue slots or the number of functional units (FUs) in the datapath in order to provide a desired throughput of one instruction per cycle. Realistically, in an embedded VLIW processors like TI C6x series from Texas Instruments [TI00], this width would be about 256 bits (eight FUs with 32 bit operations). Even if the L0 buffers store compressed instructions (NOP Compression or more advanced instruction encoding), the width of the buffer still needs to be as wide as the uncompressed case in order to provide the necessary best case throughput. With an increase in number of FUs the width of the wordlines is bound to increase. In general, memories with wide wordlines tend to be energy inefficient. Partitioning or sub-banking is a known technique to avoid long wordlines. However, these techniques are applied at the $microarchitectural$ level or at the hardware level. In contrast we propose to $raise$ the notion of partitioning to the architectural level where certain features of the application can be exploited to achieve higher energy efficiency. Since we expose the partitions at the architectural level, necessary extensions to the local controllers have to be made. In the next section, we propose two schemes for the local controllers. In this chapter, that control is however limited to conditions and the loop nest still has to be fully compatible between the L0 buffers, i.e. the different nests should have corresponding iterator positions and ranges. That limitation will be lifted in the next chapter.

The rest of this chapter is organized as follows. Section 5.2 discusses the distributed L0 buffer concept. Section 5.3 presents an illustration. Section 5.4 provides the architecture level evaluation. Section 5.5 discusses the related work and provides a comparison. Section 5.6 combines the L0 and data cluster concepts, and presents the practical consequences. Finally, Section 5.7 concludes this chapter and summarizes the key messages of this chapter. For simulation results presented in Section 5.4.2 in this chapter, the cluster generation tool presented in Chapter 5 of [Jay05a] has been used.

5.2 Distributed L0 buffer organization

The essentials of the proposed distributed L0 buffer organization are illustrated in Figure 5.1. The L0 buffers are partitioned and grouped with certain FUs in the datapath to form an instruction cluster or an L0 cluster. In each

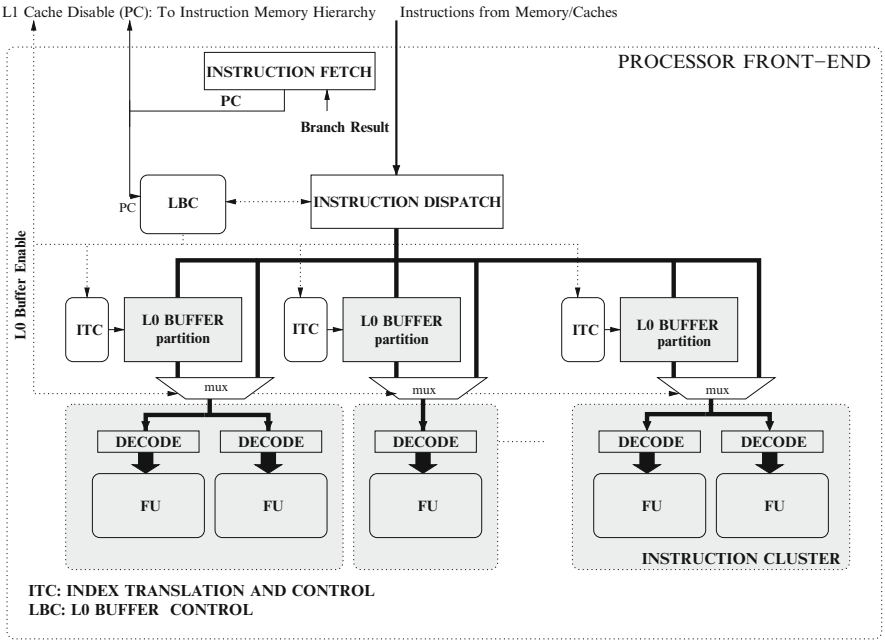


Figure 5.1: The distributed L0 buffer organization with local controllers that enable distinct conditions but a single loop nest organisation

cluster the buffers store only the operations of a certain loop destined to the FUs in that cluster. Furthermore, the buffers are placed close to the FUs. By closeness, it is meant that the latency of transfer of the instructions from the buffers to the FUs is minimal and also the physical distances between the buffers and FUs in a cluster is as small as possible.

The operation of distributed L0 organization is as follows. By default the L0 buffers are not accessed during the normal phase of the execution, which in turn results in a potentially significant reduction in the energy consumption. In Figure 5.1 one can notice that operations are loaded from the L1 in non-loop mode and from L0 only in loop mode. Parts of the program that are to be fetched from L0 buffers should be marked explicitly either by the programmer or the compiler. A special instruction $lbon$ (loop buffer on) should be inserted at the beginning of the program segment along with the number of instructions in the program segment. The program segment can be any loop with $conditionalconstructs$, $nested\ loop$ or even $parts$ of several loops. By arranging the code in a proper layout, any generic program segment can be mapped. For our analysis, we have chosen loops that have significant weight in the program execution. An example illustrating this process is shown in

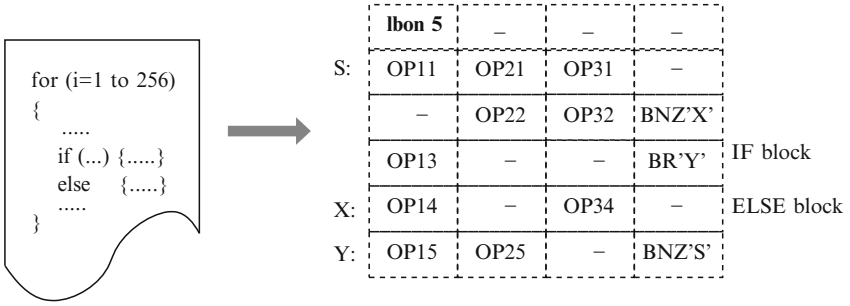


Figure 5.2: A part of the program segment mapped onto the L0 buffers

Figure 5.2. Here, a loop is explicitly marked by the compiler to be mapped onto the L0 buffers, and also the number of instructions in the loop (five instructions) is indicated.

5.2.1 Filling distributed L0 buffers

Once the instruction containing the *lbon* operation is encountered during the program execution, the processor pipeline is stalled and the instructions that immediately follow *lbon* are fetched and distributed over the different L0 partitions. The number of instructions pre-fetched will be as indicated in the *lbon* operation (five instructions in the example illustrated). Alternatively, clever pre-fetching schemes can be adopted in order to avoid the stalls (e.g. [Jou90]). For every instruction fetched, the instruction dispatch stage issues the operations to their corresponding clusters. Once the instructions are stored in the L0 buffers, the execution is resumed with instructions now being fetched from L0 buffers. The dispatch logic does not decode the operations, but partially decodes the instructions to extract operations for each cluster. Here, we assume that this logic is negligible and it is not considered in further analysis and discussions. Additionally, the buffers can also be used to store decoded operations. However, this decision requires analysis of the instruction encoding and the trade-off between sizes of L0 buffers before and after decoding. It is highly desirable that the L1 instruction memory is put in low-power (leakage reduction) mode whenever it is inactive for the duration of the inner loop iterations.

Alternatively, the L0 buffers can be filled with the instructions of the loop by simultaneously feeding the datapath during the first iteration of the loop, thus avoiding the stall cycles. However, this alternative is suitable only for loops without conditional constructs. For a loop with conditional constructs, some of the basic blocks may not be executed in the first iteration. In the worst case, one of the basic blocks may not be executed until the last

iteration. In this scenario instructions would still be fetched from expensive L1 cache instead of L0 buffers. However, this can be solved to some extent by employing code transformation techniques like function in-lining [Liv02], loop splitting, loop peeling and code hoisting [Sia01].

5.2.2 Regulating access

One of the key features of our distributed organization is that we can restrict the accesses to partitions that are not active in an instruction cycle. We achieve this by providing an *activation trace* (AT) in the local controller (ITC) of each cluster. While operations of each instruction in the loop are fetched and distributed among the partitions, a one or a zero is stored in the activation trace register indicating that the partition is active or inactive respectively. Figure 5.3 shows the activation trace for the example illustrated in Figure 5.2. For instance, during the execution of the third instruction of the loop, partitions one and four are active while partitions two and three are inactive. Thanks to this activation trace we can now restrict the access to partitions two and three through the 'enable' signal, thus saving energy.

Energy saved by *switching off* an L0 buffer partition (the L0 buffer plus its functional units) is higher than regulating accesses per cycle [Bur92]. In the proposed organization, we can detect if a particular L0 buffer partition is not used. If the activation trace is filled with zeros, then the corresponding partition is not used and hence it can be switched off. Alternatively, if the counter (described in the following section) that keeps track of the next free location in the L0 partition is zero, then we know that the corresponding partition is not used. These two detection schemes employ some hardware logic. The compiler could also indicate if a partition is not used. This requires the compiler to be aware of L0 clusters in the processors. By the extensions described in Chapter 4 of [Jay05a], the compiler can detect if an L0 cluster is used. By using any of these detection schemes L0 buffer partitions can potentially be switched off to save energy.

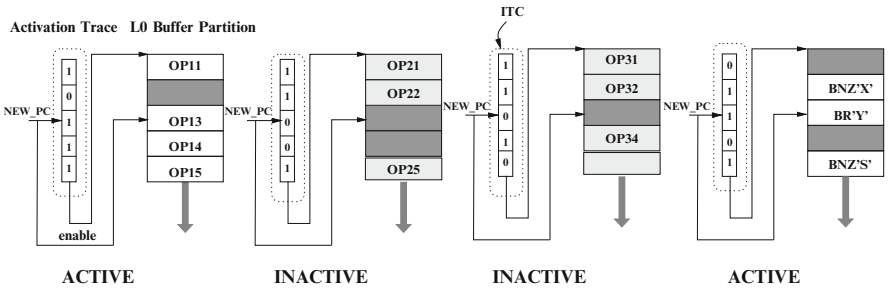


Figure 5.3: L0 buffer operation with activation based control scheme

5.2.3 Indexing into L0 buffer partitions

In order to store and fetch instructions, indexes that point to appropriate locations in each L0 partition have to be generated. One of the following two schemes can be adopted for the index generation. In the first scheme, a common index (NEW_PC in Figure 5.3) is generated for all the L0 partitions. This index is derived directly from the program counter as described in Eq. 5.1

$$NEW\_PC = fn(PC,\ START\_ADDRESS) \tag{5.1}$$

Having only one index for all the L0 partitions, implies that the operations of an instruction that are stored in different partitions have to be stored in identical locations in the corresponding cluster. For instance, the third instruction of the example illustrated in Figure 5.2 has two operations op13 and BR'Y'. These are stored in L0 partitions one and four at location two. Although only two operations are stored, the corresponding locations in L0 partitions: two and three, cannot be reused to store operations of other instructions. Furthermore, this also implies that the number of words in each partition has to be identical. One of the advantages of this scheme is that the index generation is simple but this comes at the expense of inefficient storage utilization.

In the second scheme, instead of only one index for all the partitions, separate indexes for each L0 partition are generated and stored in an Index Translation Table (ITT) as shown in Figure 5.4. Here, a counter keeps track of the next free location available in each partition, and this is incremented only when an operation is stored in that partition. Furthermore, all the ITTs are in turn indexed by the NEW_PC, which is generated as described above. The operation of this indexing scheme is illustrated in Figure 5.4. For instance, the operations of the third instruction in the above example, are stored in locations one in the first partition and one in the fourth partition, while nothing is stored in partitions two and three, thus utilizing the storage space more efficiently than the first scheme. However, this efficiency comes at the expense of increased complexity and cost of index translation in each

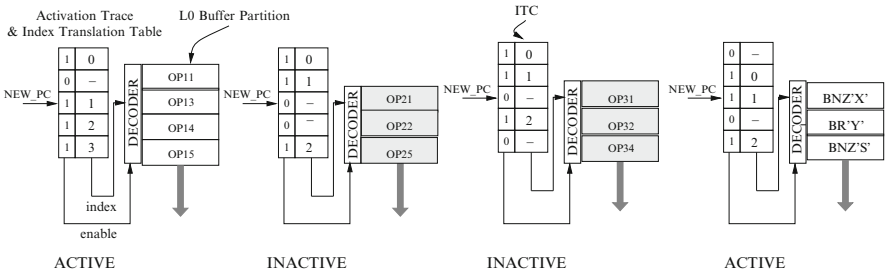


Figure 5.4: L0 buffer operation with activation based control along with index translation

partition. Unlike the previous scheme where only one index is used for all the partitions, the local controller in this scheme requires a storage for the index translation of width and depth as shown in Eqs. 5.2 and 5.3 respectively.

$$width = \log_2(Depth_{L0_i}) \tag{5.2}$$

$$depth = \max(N_{instructions\ mapped}) \tag{5.3}$$

where,
$Depth_{L0_i}$:= # of entries in L0 partition i, and
$\max(N_{instruction\ mapped})$:= maximum # of instructions among all the blocks that are mapped onto the L0 buffers.

5.2.4 Fetching from L0 buffers or L1 cache

When the $lbon$ instruction is encountered during execution, the address location of the first instruction of the loop and the address of the last instruction of the loop are stored in the start and end registers provided in the Loop Buffer Control or the LBC (not shown in the figure). When the program counter points to a location within this address range, the instructions are fetched from the L0 buffers instead of the L1 cache. The signal $L0\ buffer\ enable$ (or $L1\ Cache\ Disable$) in Figure 5.1, selects the appropriate inputs of the multiplexers and enables or disables the fetch from L1 cache.

The start register is comparable to a tag in conventional caches. Typically, when the instruction $lbon$ is encountered during execution, the start address of the loop body following that instruction is compared with the start address already stored in the start register. If there is a match, then the instructions that are already stored in L0 buffers are used. On the other hand if there is a mismatch, only then the instructions of the loop body following the $lbon$ instruction are fetched and stored in the buffers. This prevents unnecessary fetch of the same instructions and hence reduce accesses to energy expensive L1 Cache.

For the above example (Figure 5.2), a detailed illustration of the operation of distributed L0 buffers with two schemes of controller is provided in the following section.

5.3 An illustration

Figure 5.5 illustrates the distributed L0 buffer operation with activation trace and without ITT. For simplicity each FU is assumed to have a separate L0 buffer partition. A sample loop and its corresponding schedule is shown at the top of the figure.

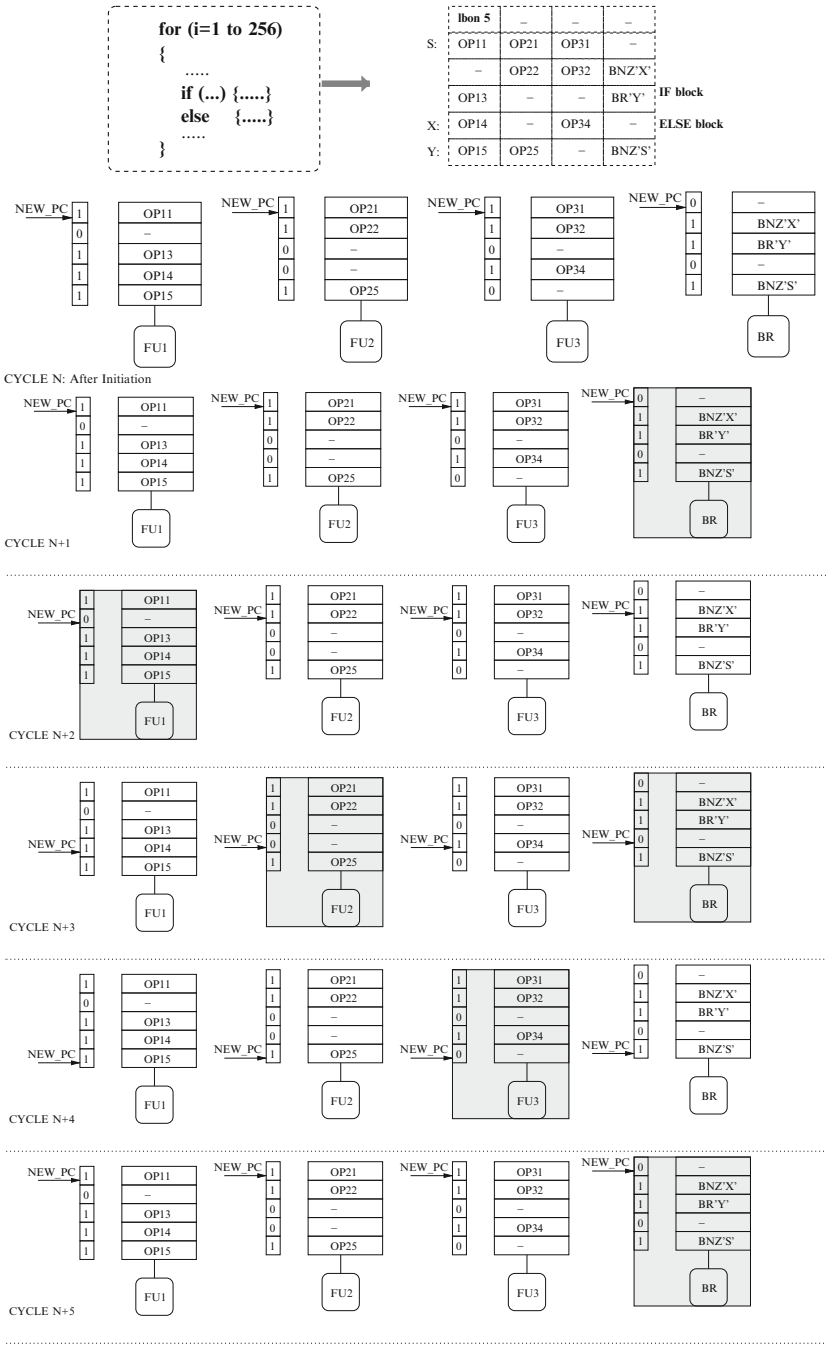


Figure 5.5: An illustration of distributed L0 buffer operation with *Activation Trace* and without *Index Translation Table*

When the instruction *lbon* is encountered during the execution, the operations within the loop are distributed into the corresponding clusters. After this initiation, at the end of CYCLE N, the operations are now fetched from the L0 buffers. During the first cycle (refer stage CYCLE N+1 in Figure 5.5) of the loop, the NEW_PC indexes into the activation trace. If a '1' is stored at that index, the corresponding L0 buffer is accessed. During this cycle the fourth cluster is not accessed. During the second cycle (refer stage CYCLE N+2 in Figure 5.5), the first cluster is not accessed. Additionally, a branch is encountered BNZ'X'. Assuming that the result of the branch is available within one cycle and the result indicates the branch to be taken, then the NEW_PC is updated so that in the next cycle it points to the appropriate instruction. During the third cycle (refer stage CYCLE N+3 in Figure 5.5), the NEW_PC points to the fourth instruction of the loop and executes the appropriate operations. During the fourth cycle (refer stage CYCLE N+4 in Figure 5.5), the operations of the last instruction are executed. If the execution is not in the last iteration then the branch points to the first instruction and the execution continues with instructions being fetched from L0 buffers. If the execution is in the last iteration, then the branch points to a location out of the address range of the loop and the instructions are now fetched from L1 cache.

Figure 5.6 illustrates the distributed L0 buffer operation with activation trace and index translation table. The execution is similar to the scheme illustrated in Figure 5.5, but for the two main differences. First, the sizes of the L0 buffers are optimized according to the active operations in the loop. Secondly, the NEW_PC indexes into activation trace and an index translation table. For a certain NEW_PC, the index stored in the translation table points to the exact location of the operation to be executed.

5.4 Architectural evaluation

For our evaluation to be realistic we have modeled the L0 buffer organization based on a known embedded VLIW processor from the TI C6x processor series [TI00], with eight FUs (eight issue slots) and an instruction width of 256 bits with 32-bit operations for each FU. Using the compiler and simulator of the framework presented in the previous chapter, applications were mapped onto this processor model and simulated to generate the profiles. The compiler in particular has been extended to identify loops which have less than 512 operations (64 instructions) and which have significant weight in the execution time, to be mapped onto the L0 buffers [Vda03].

Since our domain of interest is embedded multimedia applications, we have chosen the benchmarks for our evaluation from Mediabench [MedB]. Some characteristics of these benchmarks are shown in Table 5.1.

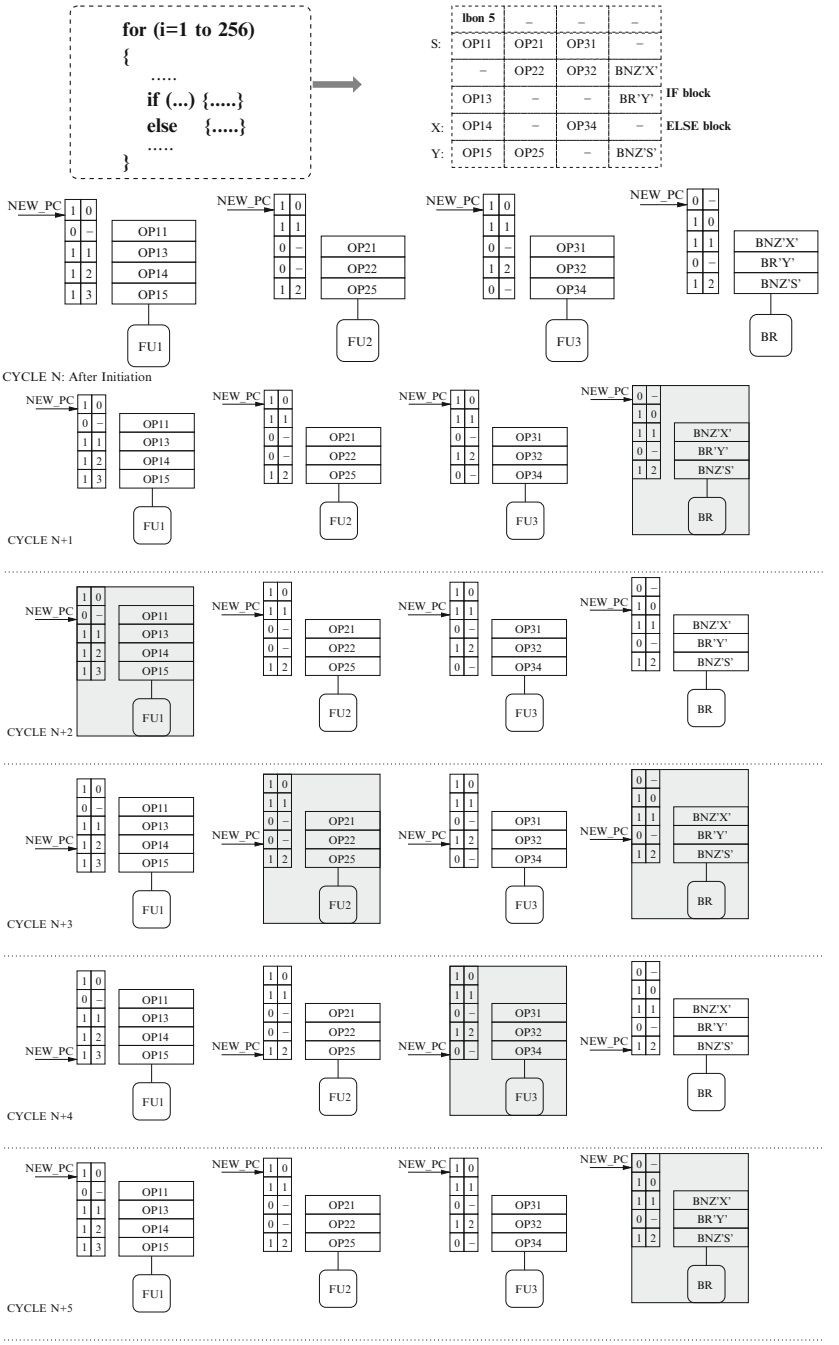


Figure 5.6: An illustration of distributed L0 buffer operation with *Activation Trace* and *Index Translation Table*

Benchmark	Type	Avg ILP (in loops)	Max size inst's (ops)
adpcm dec	Audio decoding	4.10	22 (176 ops)
adpcm enc	Audio encoding	2.65	22 (176 ops)
cavity detector	image processing	1.50	26 (208 ops)
g721 dec	Audio decoding	4.10	12 (96 ops)
g721 enc	Audio encoding	4.00	51 (408 ops)
gsm	Wireless	2.75	55 (440 ops)
jpeg enc	Image encoding	2.38	38 (304 ops)
mpeg2 dec	Video decoding	2.80	47 (376 ops)

Table 5.1: Characteristics of the benchmarks

The energy consumption of the L0 buffers and the local controllers is represented by the Eq. 5.4. This equation is based on the energy model described in Appendix B of [Jay05a].

$$E = \sum_{i=1}^{N_{clusters}} (E_i * N_i + LC_i) \tag{5.4}$$

where, E_i is the energy consumed for any random access, N_i is the number of accesses made during the program execution and LC_i is the local controller energy per cluster. For all L0 buffers and the local controllers, the E_i are obtained by modeling them as single read, single write port register files in Wattch [Bro00b] in a 0.18 µm technology.

5.4.1 Energy reduction due to clustering

Clustering the storage at an architectural level aids in reducing the energy consumption in two ways. First, smaller and distributed memories can be employed. Second, at the architectural level an explicit control over the accesses to these memories can be imposed (through the local controller).

As described in Section 5.2, with the aid of ITT the depths of L0 partitions can be optimized independently in each partition. This corresponds to reduction in effective buffer energy per access (ΣE_i). Figure 5.7 shows the reduction in effective buffer energy per access[1] for increasing number of clusters. For instance when the number of clusters is equal to four, the effective buffer energy per accesses is reduced by about 20%. We see that with an increase

[1]For a single cluster, the energy for AT+ITT is slightly more than the energy for AT. This difference is due to additional address decoder used for the buffer, instead of one-hot encoding.

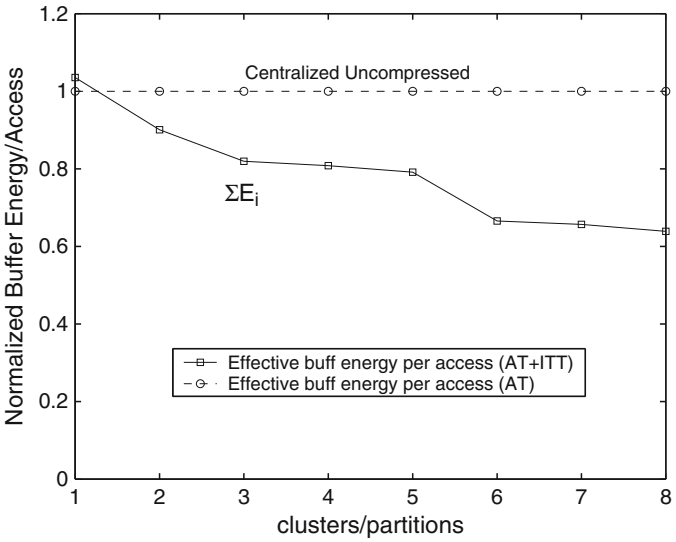


Figure 5.7: Reduction of effective buffer energy per access (ΣE_i) due to index translation table (ITT) with increasing number of clusters ($N_{clusters}$)

in number of clusters the effective buffer energy per access reduces, and it is minimal when number of clusters is equal to number of functional units. For this analysis we have assumed that clusters are generated randomly. However, as demonstrated in Chapter 5 of [Jay05a], by generating clusters with the knowledge of memory access patterns, the optimal number of clusters is not 8, as implied in Figure 5.7, but somewhere in between 1 and 8.

By restricting the accesses to the buffers we can reduce the amount of switching energy in the L0 buffers. Figure 5.8 shows the reduction in the effective number of accesses (ΣN_i) with increase in number of clusters. Here, effective number of accesses is defined as sum of all the accesses per functional unit.[2] We see that with increase in number of clusters, the effective accesses keep on reducing and are minimal when number of clusters is equal to number of functional units. The reduction is fairly intuitive because with increase in number of clusters, the degree of control over effective number of accesses per functional unit increases and when each functional unit has its own buffer partition, this degree is maximal.

The aforementioned reductions reduce the buffer energy. However, this reduction is traded-off against the increase in local controller energy. Figure 5.9 summarizes the trade-off between the buffer energy ($\Sigma E_i * N_i$) and the local

[2]Here, effective number of accesses is defined as: $\sum_{i=1}^{N_{clusters}}(N_i * NFU_i)/N_{FU}$. Where, $N_{clusters}$ is the number of L0 clusters, N_i is the number of accesses for cluster i, NFU_i is the number of FUs in cluster i and N_{FU} is the total number of FUs in the datapath.

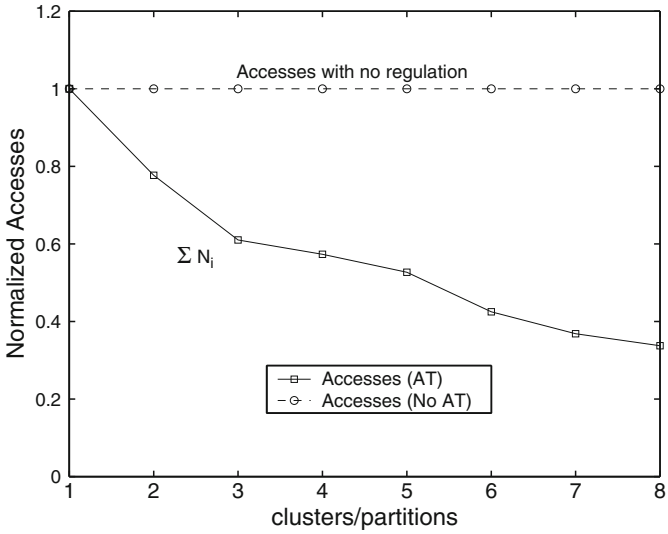


Figure 5.8: Reduction in effective number of accesses (ΣN_i) due to activation trace (AT) with increasing number of clusters ($N_{clusters}$)

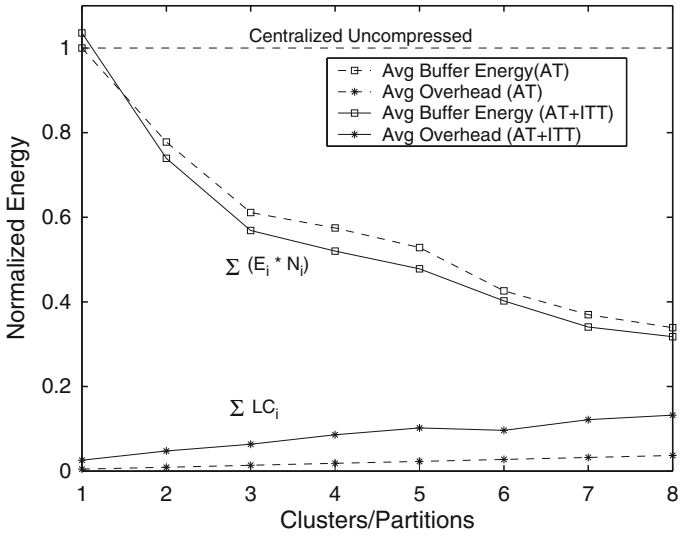


Figure 5.9: Reduction in buffer energy and increase in local controller energy

controller (ΣLC_i) for the two proposed schemes and Figure 5.10 shows the total energy reduction for the two schemes. Since for the scheme represented by Figure 5.4, buffer energy is reduced due to both regulating the accesses and reducing the effective size, this reduction is greater than the energy

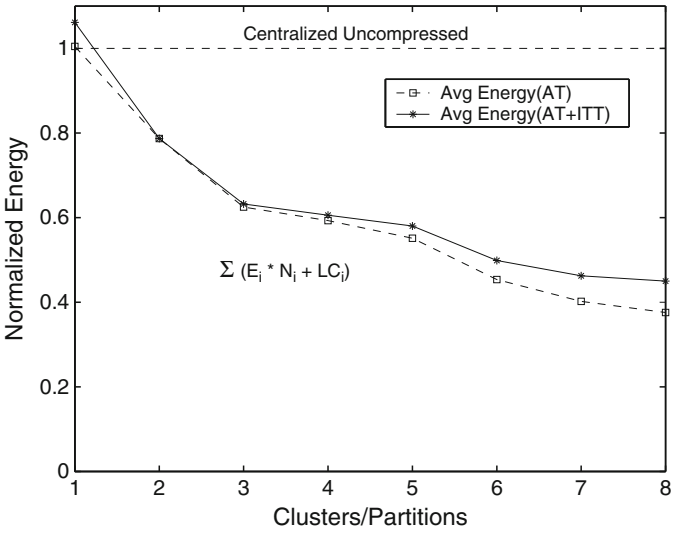


Figure 5.10: Reduction in total energy for two L0 buffer schemes

reduced for the scheme represented by Figure 5.3, where only accesses are regulated. As expected, the local controller energy in the former is larger than the local controller energy in the latter due to increased complexity. However, Figure 5.10 shows that in some cases increased complexity in the local controller pays-off against reductions in buffer energy.

5.4.2 Proposed organization versus centralized organizations

We have evaluated two centralized L0 buffer schemes, namely a centralized uncompressed scheme and a centralized compressed scheme, against our proposed organizations, a distributed L0 buffer with activation trace and a distributed L0 buffer with activation trace and index translation. Figure 5.11 summarizes the energy reductions of various schemes on different applications of the Mediabench [MedB] suite. On average the energy consumption in the proposed distributed organization is about 63% lower than the energy consumed in an uncompressed centralized scheme, and about 35% lower than the energy consumed in a centralized compressed scheme.

For the centralized uncompressed scheme the size of the L0 buffer, for each application, will be the maximum number of instructions among all the loops that were identified by the compiler. However Table 5.1 indicates that the average ILP is typically less than the width of eight operations per word in L0 buffer, and hence the L0 buffer is unnecessarily large and energy inefficient.

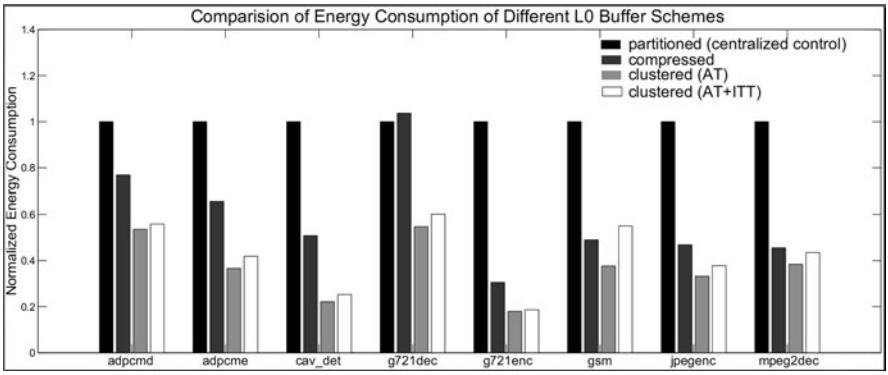


Figure 5.11: Energy consumption of distributed organization in comparison with other schemes

In contrast, a centralized compressed L0 buffer efficiently utilizes the storage and the depth of the L0 buffer can be made smaller. For the benchmark 'mpeg2dec', we observed that the depth of the L0 buffer could be reduced from 47 to 18. This reduction comes from the fact that the instructions are of variable length and the operations in an instruction are tightly packed eliminating the NOPs. Here, we have adopted the instruction fetch model from the TI C6x processor series, where every fetch to the L0 buffer partition fetches an instruction packet of eight operations. This packet is stored in an additional buffer and the operations are fed to the datapath from this buffer every instruction cycle. A new instruction packet is fetched only when operations in the additional buffers are used up. Based on this model, we can see that on average 44% of energy can be reduced over an uncompressed centralized scheme. The number of fetches to the L0 partition is reduced significantly but at the expense of adding an additional buffer. However, in most cases this overhead is compensated by the reductions in the L0 buffer except for one particular benchmark, g721dec. For this benchmark the energy reduction in the L0 buffer (reduction in depth) was not sufficient to compensate for the overhead (refer Figure 5.11).

In the distributed scheme with both AT and ITT as opposed to distributed scheme with AT only, in addition to reducing the number of accesses in each partition, the depths of the L0 buffers in each partition can be further optimized. This reduction in the L0 buffer size comes at the expense of increased complexity and energy consumption of the controller. However, this increase in energy is just large enough not to be compensated by the reduction in the L0 buffer energy. Figure 5.11 shows that the energy consumption of the distributed organization as proposed in Figure 5.4 (local controller with both Activation Trace and Index Translation Table) is slightly more than the energy consumption of the distributed organization proposed

in Figure 5.3 (local controller with only Activation Trace). In our analysis of the distributed organizations we have assumed that only one type of local controller is used throughout. However, a hybrid scheme could also be employed where in some clusters have activation trace while the others have both activation trace and an index translation table. Currently, we have not made any analysis regarding the hybrid scheme, and we leave such an analysis for future work.

5.4.3 Performance issues

From Section 5.2.4, we can deduce that the number of cycles lost by stalls due to fetching depends on the number of instructions in the loops that are mapped to the L0 buffers. However, in comparison with the number of cycles the instructions in the loops are executed, the stall cycles are negligible. From Figure 5.12 we can observe that the performance degradation due to prefetching is less than 5%.

In the distributed organization as shown in Figure 5.4, two storage blocks have to be accessed sequentially in one cycle, namely the Instruction Translation Table and L0 buffer partition. While this requirement may seem to constrain the cycle time, in reality it does not. In embedded processors which operate at low frequencies (100–1,000 MHz), two storage blocks can be accessed within one instruction cycle. For instance, in the benchmark 'gsm', the L0 buffer size of one partition is about 3kb and the corresponding size of the local controller is about 0.5kb. For the register file model in (0.18 µm technology), the access times to the buffer and the controller are about 2.5 and 2.0 ns respectively. Together, the critical-path length is about 4.5 ns, translating to about 250 MHz, which is about the same as the operating frequency of

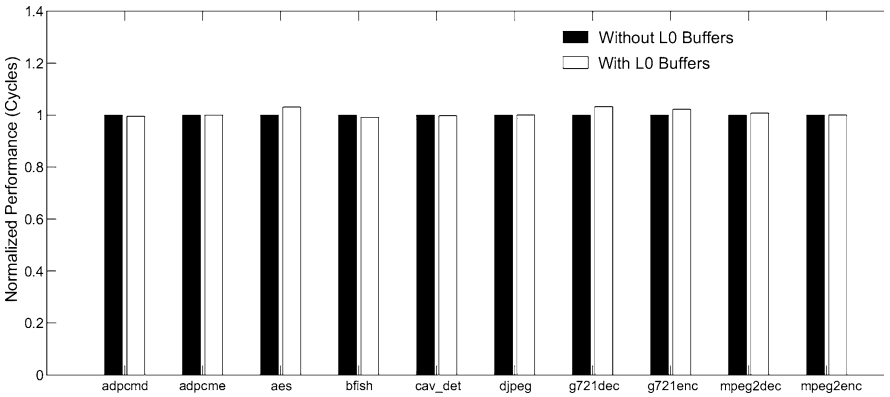


Figure 5.12: Performance degradation due to filling in L0 buffers

some of the TIC6x processor series in 0.15 µm technology [TI99]. However, even if the access times are not within the critical path length of the processor, the L0 buffer access in the proposed scheme can be *pipelined*. In the first stage, the local controller can be accessed to get the activation and the index, while in the second stage the operations stored in the buffer can be retrieved.

5.5 Comparison to related work

The available literature for instruction memory organization falls under two broad categories. The first category encompasses the literature available in relation to L0 buffers or loop buffers, which is one of the central concepts of our proposed organization. We give an overview of different flavours of L0 buffer organization and indicate that our approach is complementary to most of them. The second category encompasses the literature available in relation to partitioned or decentralized organization. We give an overview of different partitioned organizations especially in relation to the instruction memory and the processor front end.

The concept of using small buffers has been applied to optimize both performance and energy. Jouppi [Jou90] has studied the performance advantages of small pre-fetch buffers or stream buffers. On the other hand, the reduction of energy using small buffers was first observed by Bunda [Bun93], and this idea was more generalized as Filter cache [Kin00] by Kin et al. The authors have shown that up to 58% of instruction memory power can be reduced with a performance degradation of about 21%. To mitigate the loss in performance, Tang et al. [Tan01] have proposed a hardware predictive filter cache. Alternatively, authors in [Lee99, Baj97, And00], proposed to use these buffers only for loops, thus reducing the loss in performance while still retaining the large reductions in energy.

Since the identification of loops to be mapped onto the L0 buffers is largely hardware controlled and dynamic, loops with small iteration count could also be mapped onto the L0 buffer leading to thrashing. Vahid et al. [Gor02b] have analyzed this situation and they propose a pre-loaded loop cache, where the loops with large instruction counts are identified by profiling and only these are mapped onto the loop cache. Furthermore, their scheme also support loops with control constructs and various levels of nesting.

In a partitioned organization [Shi99, Con96a], a buffer is divided into smaller partitions in order to reduce the wordline width. However, the process of partitioning is largely arbitrary. The operations of a certain functional unit are not necessarily bound to few partitions, they can be placed in any of the partitions. Thus, no correlation exists between the process of partitioning and the functional unit activity. A correlation between the two should be

explicitly imposed in order to physically place the partitions over different functional units in the datapath and ease the constraints on the interconnect. Otherwise, an operation for a functional unit may need to be fetched from a partition which is physically placed close to a different functional unit, and thus constraining the interconnect severely. In this sense, we follow a partitioning or clustering scheme which is different and at a higher level of abstraction than the conventional partitioning scheme.

Similar to the distributed organization presented in this book, small loop buffers are present in the scalable dual-fetch architecture [Bar05a]. In that, a master fetch path consists of an level-1 I-cache and a loop buffer. The slave paths consists of several clusters with loop buffers in each, where the instructions are fed directly from a level-2 memory. The key differences with the dual-fetch architecture are as follows.

In the dual-fetch architecture [Bar05a], the instruction (or L0) clusters and data clusters are identical, i.e., the FUs comprising within a data cluster and L0 cluster are the same. However, a clear distinction between instruction and data cluster is made in this book and the the FU composition can be largely independent. Additionally, since instruction and data clusters are identical in the dual-fetch architecture assignment options for instruction clusters are limited. Additionally, operation assignment scheme is primarily directed at data clusters. In this book however, we propose that an explicit stage is added (in addition to data clustering) to perform scheduling and assignment for L0 clusters (see also Chapter 4 of [Jay05a]).

In the dual-fetch architecture, the slave paths are primarily controlled by the master path through a special dedicated operation in the instruction format. The opcode fields of that operation corresponds to slave clusters which provide a mechanism for indexing and access regulation. However in this book, the instruction clusters are controlled in a distributed manner with local controllers providing the indexing and access regulation mechanisms. Additionally, the indexing in dual-fetch architecture is centralized with one index per cycle for all the clusters. However, an $index\ translation$ scheme is provided in this book that enables the L0 clusters to have more decoupled L0 buffers, i.e., the size of an L0 buffer in a cluster is independent of other clusters. By combining indexing of dual-fetch architecture with the index translation of this book, the two fetch mechanisms can be made compatible.

At a conceptual level, the distributed L0 buffer organization proposed in this book is similar to an n-way associative cache with way-prediction [Pea01] or even horizontally partitioned caches with cache-prediction [Kim01]. In associative caches, instructions are stored in different 'ways', while in our case, the instructions are stored in different clusters. The way-predictors predict the 'ways' that are to be accessed in any instruction cycle, while in our case the local controllers regulate the accesses to each cluster. In spite of

these similarities, the underlying details of associative caches and distributed L0 buffers are different. Firstly, the way prediction schemes are much more complex than the local controller schemes proposed in this book, and they still rely on tags for addressing. Secondly, most of the way prediction schemes have been applied in the context of hardware controlled caches, and thus there is a possibility of mis-prediction. However, in our case the L0 buffers are software mapped and hence the activation of each partition can be known beforehand, thus avoiding any mis-prediction.

Superscalar processors are high-performance (GHz $range$) and high power consuming (10–100 W) desktop oriented processors. On the other hand, embedded processors are high-performance (100 Mhz $range$) and at least two to three orders of magnitude lower power consuming (0.1–2 W). Their processor characteristics vary significantly [Fly99]. However, the notion of clustering is also seen in some of the superscalar processors. Here, we mention only a few distributed (decentralized) architectures. Zyuban et.al. [Zyu01] have analyzed the effects of clustering the front-end of a superscalar processor, particularly on energy. They also propose a complexity-effective multi-cluster architecture that is inherently energy efficient. Many other research groups have proposed some form of decentralized organizations [Fra93, Pal97]. However, their primary concern was mainly performance and not energy costs.

5.6 Combining L0 instruction and data clusters

In the previous sections, instruction memory hierarchy issues were discussed independently of data memory hierarchy. The results presented earlier, assume a centralized data memory hierarchy, i.e., a single data memory architectural block at different levels of memory hierarchy. This includes single (but multi-ported) register file, per processor, at the lowest level, single level 1 data cache and an external memory. However, in realistic systems foreground data memories are also clustered. For instance, in TMS320C6000, a VLIW processor from Texas Instruments has two register files at the lowest level of memory hierarchy. Systems designed by following the DTSE methodology often have multiple level 1 data memories. Since the execution of instructions depend on the data, instruction and data memory organizations affect each other to various degrees depending on the system abstraction level and the memory hierarchy level. From the work in [Vda05], we can learn that the data memory optimizations at the software and architectural level for level 1 and higher levels of memory hierarchy, have considerable impact on the instruction memory organization in terms of energy.

In the remainder of this chapter we show that these problems can be constraint orthogonalized by optimizing for data first, followed by propagating

the resulting constraints and then optimizing for instructions. We can still achieve significant energy efficiency in L0 buffers, where the optimum is close to the optimum when L0 clusters are generated for a centralized data register file. Furthermore, the optimizations for instructions are applied per data cluster. We follow this approach at the lowest level of memory hierarchy (L0 buffers and register files) and at the architectural level of system abstraction.

As mentioned before at the architectural level of system abstraction, both instruction memory and data memory are organized in levels. These levels are conventionally distinguished by difference in the access times: as measured in number of cycles per random memory access. Higher levels of memories typically have higher access times. Alternatively, multiple levels can also be distinguished by the physical distances from the data path (or functional units). Register files and L1 data caches might have the same access times, but register files are closer to the datapath, hence register files can be considered to be at a lower level in the memory data memory hierarchy.

In a datapath cluster, the functional units derive data from a single register file. In contrast, the functional units in an L0 cluster derive instructions from a single L0 buffer partition. If register files can be viewed as the lowest level storage in the data memory hierarchy, then the L0 buffers can be viewed as the lowest level of storage in the instruction memory hierarchy. Some clustered VLIW architectures proposed in literature, in particular the Lx [Far00] processor, have a notion of instruction cluster control similar to the L0 clusters (except for the explicit introduction of the copies to the L0 buffers). However in their architecture, a datapath cluster and an instruction cluster are fully equivalent. In contrast, we distinguish between the two explicitly. For instance, we could employ several L0 clusters within a datapath cluster (or vice versa). That will be worked out in more detail and illustrated in the rest of this section.

In terms of number of accesses, the cumulative number of access to register files are higher compared to L0 buffers. In most datapath organizations for each execution of an operation, two read accesses and one write access to the data register files is needed, while only one read access to the instruction memory (L0 buffer) is needed. Additionally, for VLIW organizations the register files are multi-ported. This implies that for every operation, there are three accesses to an energy expensive data register file and one access to a relatively energy inexpensive L0 buffer.

5.6.1 Data clustering

In conventional VLIWs large multi-ported register files are a major bottleneck for energy efficiency. Various clustered organizatons have been proposed to

increase the energy efficiency of the register files [Rix00a]. Reducing register file energy by clustering, results in performance loss. By clustering, the number of ports per register file is reduced and hence the energy. However, since the data is distributed across the clusters, the data produced in one cluster may be needed in another cluster. This inter-cluster data communication is an overhead, since a number of cycles and operations are wasted in transferring data from one cluster to another. With smaller data clusters, i.e. smaller number of FUs and smaller number of ports per register file, the inter-cluster communication increases and hence large performance losses are incurred. Due to this trade-off most of the state-of-the-art VLIW processors still have relatively large data clusters.

5.6.2 Data clustering followed by L0 clustering

With several optimizations to be performed at a certain system level abstraction, *ordering* these optimizations in steps is important [Cat98a]. In this particular context ordering the two optimization phases: data and L0 clustering. Data clustering phase has a larger impact in terms of energy and performance compared to L0 clustering. Also, the consequences of data clustering are larger compared to L0 clustering [Keu00, Cat99]. By consequences, we mean here the number of architectural parameters affected by the decision taken by this phase. Hence, we follow a methodology where data clustering is applied first, followed by L0 clustering.

In addition to generating L0 clusters after data clusters, L0 clusters are generated *per data cluster*. Thus we propagate the constraints of data clustering to instruction clustering. If the two phases are applied independently the resulting clusters could be misaligned as shown in Figure 5.13. Here, FUs 1, 2, 3 and 4 are in first data cluster, while FU1 is in first L0 cluster, FU2 in second L0 cluster and FUs 5 and 6 are in third L0 cluster. This misalignment in the cluster boundaries will lead to severe constraints in placement and routing of the architectural blocks. Which in turn could lead to increased energy consumption by the interconnect. In order to avoid the misalignment of cluster boundaries, L0 clusters are generated per data cluster.

In this section we show that for the current VLIW organizations, applying data clustering first followed by L0 clustering per data cluster, is a reasonable methodology. Irrespective of the data cluster generation scheme employed, by following this methodology, both types of clusters can be accommodated. And the energy reductions in L0 buffers are still substantial. An instance of the resulting clusters is shown in Figure 5.14. For instance, a datapath of eight FUs has two datapath clusters with four functional units each. This is a reasonable data cluster configuration since many state-of-the-art VLIWs, like C6x series from Texas Instruments and Lx processor series

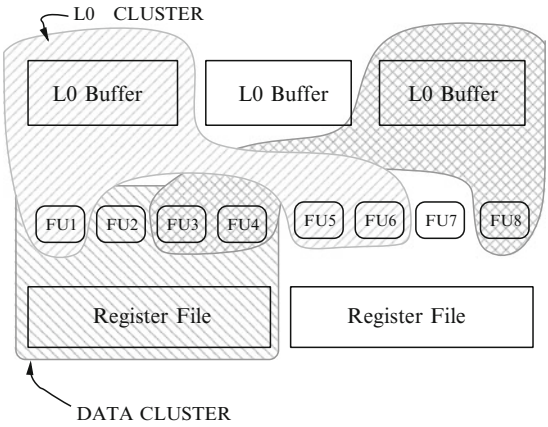


Figure 5.13: An instance where data and L0 cluster boundaries are misaligned

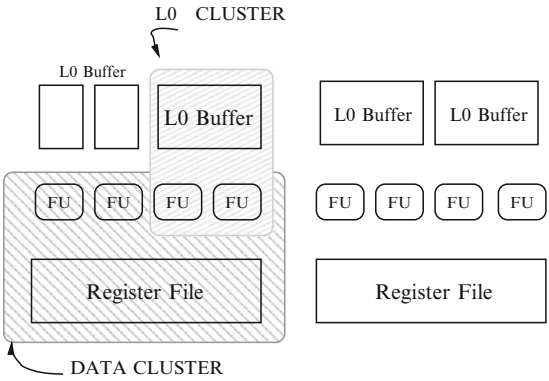


Figure 5.14: An instance when L0 clusters are encompassed in data clusters

from STMicroelectronics, have similar configurations. Additionally, within each data cluster several L0 clusters can be employed. For example, three and two L0 clusters in two data clusters as shown in Figure 5.14.

5.6.3 Simulation results

Figure 5.15 summarizes the methodology and the results of generating L0 clusters per data cluster for Mpeg2 Decoder benchmark. A datapath with two data clusters with four FUs each is assumed. This datapath configuration is modeled after the VLIW processor TMS320C6000 processor from Texas

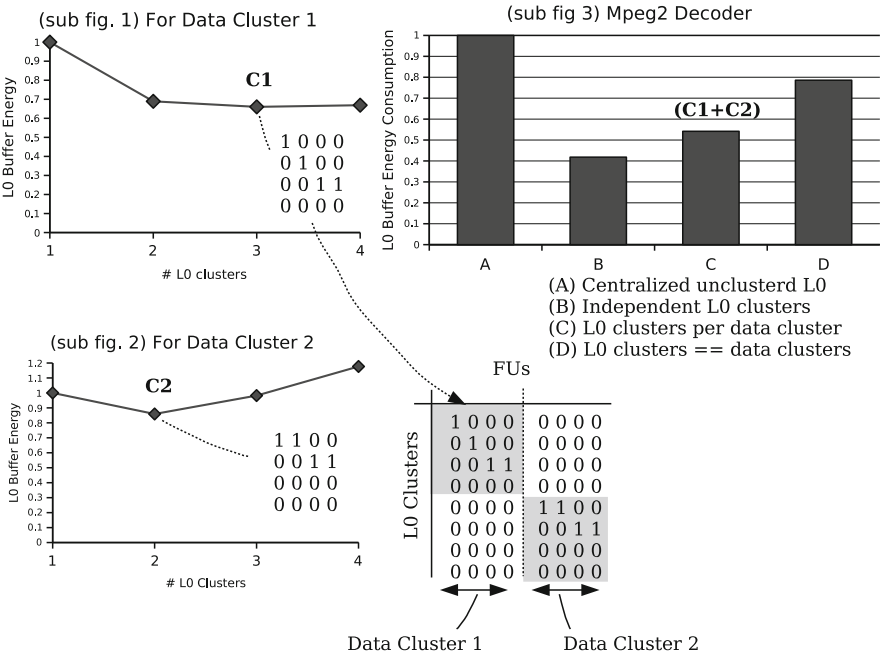


Figure 5.15: L0 cluster per data cluster as applied to Mpeg2 Decoder for a VLIW with two data clusters

FU no.	C6x	Fully Het	Homo
1	add/shft	add	add(fp)/shft/mult(fp)/ldst
2	mult/shft	shft	add(fp)/shft/mult(fp)/ldst
3	mult/add	mult	add(fp)/shft/mult(fp)/ldst
4	fp alu/ldst	fp alu	add(fp)/shft/mult(fp)/ldst
5	fp mult/add	fp mult	add(fp)/shft/mult(fp)/ldst
6	add/shft	ldst	add(fp)/shft/mult(fp)/ldst
7	mult/shft	add	add(fp)/shft/mult(fp)/ldst
8	mult/add	mult	add(fp)/shft/mult(fp)/ldst
9	fp alu/ldst	ldst	add(fp)/shft/mult(fp)/ldst
10	fp mult/add	shft	add(fp)/shft/mult(fp)/ldst

Table 5.2: Machine configurations with different FU types

Instruments. The types of FUs are as indicated in the second column of Table 5.2. All the energy figures are estimated by modeling the L0 clusters in 0.18 µm technology.

Sub-fig. 3 shows the L0 buffer energy consumption for different ways of L0 clustering. Legend (A) corresponds to centralized unclustered L0 buffer.

All the other energy figures are normalized against this number. Legend (B) corresponds to applying L0 clustering assuming that there are no data clusters. L0 clusters generated this way would yield the highest reductions (about 60% in this case), since misalignment of cluster boundaries are not considered. Legend (D) corresponds to L0 clustering, where the data clusters define the L0 clusters. No further L0 clustering is applied and the L0 clusters are equivalent to data clusters, which is, two L0 clusters with four FUs each, similar to the data clusters. Legend (C) corresponds to generating L0 clusters per data cluster. As described in Chapter 5 of [Jay05a], a systematic technique is used to generate the L0 clusters, considering the traces per data cluster. Sub-figures (1) and (2) show the result of clustering. The best L0 clustering for each data cluster, namely points C1 and C2 in sub-figures (1) and (2) respectively, are combined together to form the final L0 cluster configuration for the whole datapath. The combined configuration is shown at the bottom right of Figure 5.15.

From these results we see that, by generating L0 clustering per data cluster, the reduction in L0 buffer energy is still substantial. Namely, it is about 50% compared to centralized unclustered L0 buffer (Legend (A)), and about 30% more than by making L0 clusters identical to data clusters (Legend (D)). And, only about 10% worse than generating L0 clusters independently (Legend (B)). Figure 5.16 shows results similar to sub-figure (3) in Figure 5.15, but for all the benchmarks in Mediabench. We see that on average generating L0 clusters per data cluster is only about 10% worse than generating L0 clusters independently.

Even when the data clusters are smaller (two to three FUs per data cluster), the resulting L0 clusters by following this methodology are still energy efficient. Figure 5.17 shows the results for a datapath with three data clusters on an eight FU datapath (two data clusters with three FUs each and one data cluster with two FUs). On average, L0 clusters are only 10% worse than the L0 clusters generated independently. But also only 10% better than the

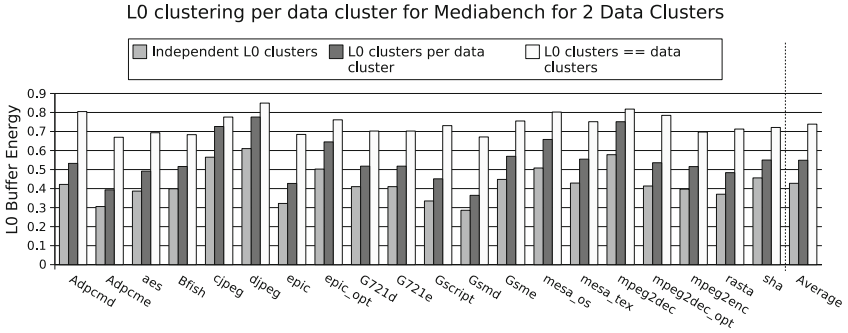


Figure 5.16: L0 buffer energy by L0 clustering per data cluster for Mediabench with two datapath clusters

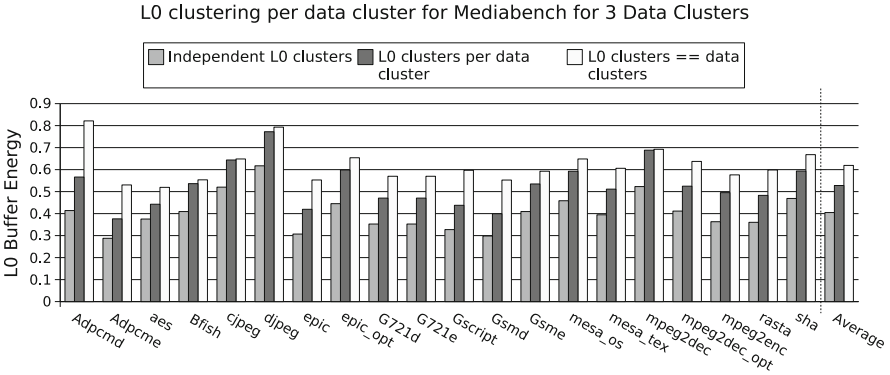


Figure 5.17: L0 buffer energy by L0 clustering per data cluster for Mediabench with three datapath clusters

L0 clusters when data and L0 clusters are identical. By making data clusters small, the freedom to generate L0 clusters is less. Hence in some cases like 'djpeg', generating L0 cluster per data clusters and making L0 clusters identical to data clusters, have similar energy figures. Typically L0 clusters generated independently are small (one to three FUs per L0 cluster). Hence by making data clusters small the difference in L0 energy, between making L0 clusters identical to data clusters and generating L0 clusters independently, is reduced. In Figure 5.16, this difference is about 30% on average, while the same difference is about 20% in Figure 5.17.

5.6.4 VLIW Variants

In some variants of VLIW architectures like Jazz processors from Improv Systems [Imp99], Transport Triggered Architectures (TTAs) [Corp98] and coarse grained reconfigurable arrays [Sin00, Col03, Ven03], the interconnect connecting the register files to FUs follow different topologies than the conventional VLIW. Those particular topologies enable the architect to employ smaller data clusters. In such cases, L0 clusters can be applied independent of data clusters without the concerns of cluster boundary misalignment. An extreme form of data clustering is shown in Figure 5.18, where each FU is a data cluster and the L0 clusters $encompass$ multiple data clusters. As a result, if the data cluster overlaps with two L0 clusters, then the L0 cluster boundary is realigned so that the boundaries do not overlap. The data cluster boundaries will not be affected. In such cases, data clustering is still applied first since they have larger impact in terms of energy and performance. However, now L0 clusters are generated independently as described in Chapter 5 of [Jay05a], with similar results.

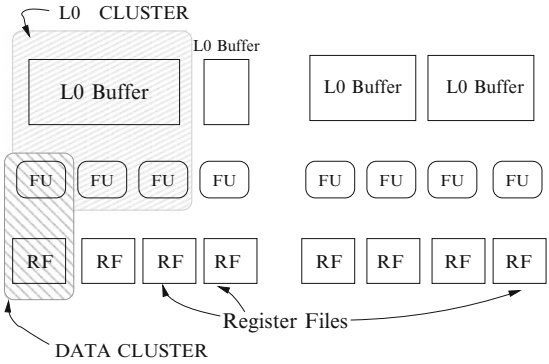


Figure 5.18: An instance when data clusters are encompassed in L0 clusters

5.7 Conclusions and key messages of this chapter

In this chapter, a distributed L0 buffer organization was presented with two different schemes to control the activation and indexing of the L0 partitions. Here, that control is however limited to conditions and the loop nest still has to be fully compatible between the L0 buffers. An analysis based on two key metrics (energy per access and number of accesses) related to energy consumption, gives an insight as to why the energy is reduced. Although Figure 5.10 indicates that a fully distributed organization is energy efficient, it has been shown in [Jay05a], that an exploration of trade-off between the local controllers and the L0 buffers yields a solution in between. The local controller with activation trace and index translation offers more freedom in the architecture (the depths of L0 buffers can be independent). In comparison to other centralized schemes energy consumption in the proposed distributed organization is about 63% lower than the energy consumed in an uncompressed centralized scheme, and about 35% lower than the energy consumed in a centralized compressed scheme. Additionally, the energy efficiency is achieved without jeopardizing performance.

Further extensions to this architecture can be conceived. Executing multiple incompatible loops in parallel has benefits both in terms of energy and performance. The organization presented in this chapter supports only the execution of a loop with single compatible threads of control. Extensions can be made to this architecture to support execution of multiple loops (multiple thread of control). This is explored in Chapter 6.

Currently the clustering is done at the lowest level of instruction storage, namely the L0 buffers. However, the notion of clustering can be extended to higher levels of instruction memory hierarchy, namely levels 1 and 2 instruction caches [Jay02b].

In addition, issues related to combining data and L0 clusters were presented. Through qualitative reasoning we proposed to order the two optimizations: data clustering $followed$ by L0 clustering. For relatively larger data clusters, to avoid cluster boundary misalignment, we propose to generate L0 clusters per data cluster. From the simulation results we can infer that by following this methodology about 50% energy saving can be obtained in the L0 buffers for Mediabench.

CHAPTER 6

Multi-threading in Uni-threaded Processor

Abstract

This chapter introduces the concept of executing multiple incompatible loops in parallel and thereby enabling multi-threading in an efficient way in a VLIW processor. The proposed multi-threading is enabled by the use of a distributed instruction memory organization with a minimal hardware overhead. This forms one of the core contributions of this book. It also shows how the proposed instruction memory hierarchy extension can both improve performance as well as reduce the energy consumption compared to state-of-the-art simultaneous multi-threaded (SMT) architectures over various DSP benchmarks. The chapter also shows that the proposed architecture can be compiled for.

6.1 Introduction

The instruction memory organization is a large energy consumer in the processor, which can become a real bottleneck after techniques like loop transformations, software controlled caches, data layout optimizations [Ban02, Kan04a] and distributed register files [Rix00a, Lap02] have been applied to lower the energy consumption of other components of the

F. Catthoor et al., *Ultra-Low Energy Domain-Specific Instruction-Set Processors*,
DOI 10.1007/978-90-481-9528-2_6,

system. This has also been shown in Chapter 4. This chapter presents a novel architectural enhancement that reduces the energy consumed in the instruction memory organization, while allowing a "virtual" multi-threaded operation inside the processor.

Previously proposed architecture enhancements aim to reduce the energy consumption of the instruction memory hierarchy for VLIW processors by using loop buffers [Jay05b], NOP compression [Hal02], SILO cache [Con96a], code-size reduction [Hal02], etc. In spite of these enhancements, the instruction memory organizations are still centralized and the resulting energy efficiency is still quite low [Jay05b]. Additionally, centralized loop buffers are not scalable to more parallel architectures.

The well known *L0 buffer* or *loop buffer* is an extra level of memory hierarchy that is used to store instructions corresponding to loops. As shown in [Jay05b] and Chapter 5, the L0 buffer organization is a good candidate for the clustered/distributed instruction memory hierarchy solution. However, the earlier described distributed loop buffer solution supports only one thread of control. In every instruction cycle, a *single* loop controller generates an index, which selects/fetches operations from the loop buffers. The loop counter/controller may be implemented in different ways: instruction based or using a separate hardware loop counter. By supporting only one thread of control different incompatible loops cannot be efficiently mapped to different distributed loop buffers (explained in detail in Section 6.2). Hence a need exists for a distributed and scalable solution which can provide multiple threads of control. That will be effectively addressed in this chapter.

Current platforms and processors need to exploit more parallelism at different levels [DeM05] to improve both performance and energy efficiency. The most important compute intensive parts of current embedded applications are written as nested loops and therefore loops form the most important platform load in a program.

On single threaded architectures, techniques like loop fusion and other loop transformations are applied to make use of the parallelism that is available within loops (boost ILP). Not all loops can be efficiently broken down into parallel operations in this manner as they may be *incompatible*. This incompatibility of loops leads to a large control overhead. Therefore different loops can be distinguished as the following types:

1. Regularity of Loop Nest
   (a) *Regular Loop Nest*: Nested loops with no data dependent conditions inside the loops
   (b) *Irregular Loop Nest*: Nested loops with data dependent conditions inside the loops

2. Compatibility of Loops

   (a) *Compatible Loops*: Two loops that have identical nesting and loop bounds. They can be merged without any conditions and have no dependencies that block merging.

   (b) *Incompatible Loops*: Two loops with different bounds and/or nesting is not identical. These types of loops need conditions to be merged into one single loop. Dependencies across the loops would also introduce extra conditions and extra iterations to be inserted, in order to align production and consumption correctly.

To be able to handle all these types of loops efficiently, multi-threaded platforms that can support the execution of multiple incompatible loops in parallel are needed. It is however essential that this can be done with *minimal hardware and instruction overhead*. This chapter proposes such a virtually multi-threaded distributed instruction memory hierarchy that can support the execution of multiple *incompatible* loops (for example, see Figure 6.1) in parallel. In addition to regular loops, irregular loops with conditional constructs and nested loops can be mapped. To make the loops fit in the loop buffers, sub-routines and function calls within the loops must be selectively inlined or optimized using other loop transformations, like code hoisting or loop splitting. Alternatively, sub-routines can be executed from

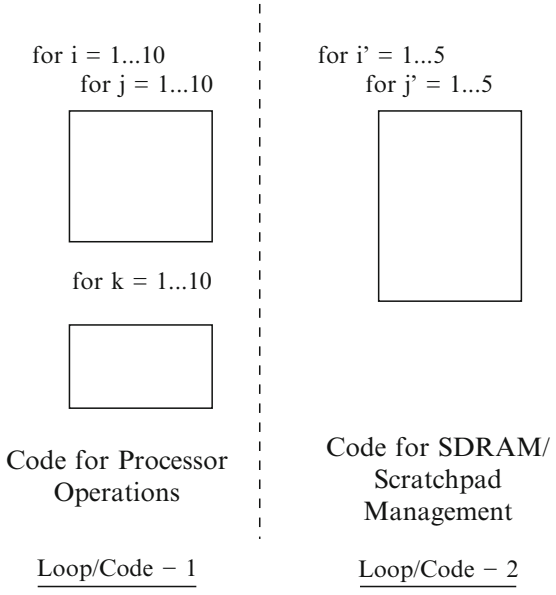


Figure 6.1: Incompatible loop organizations: a simple example

the conventional level-1 instruction cache or scratch-pad if they do not fit in the loop buffers. Another possible solution is where the sub-routine code is completely or partly loaded onto the loop buffer. This would still be a feasible solution if the gains are justified. In the proposed architecture, the loop buffers are clustered and each loop buffer has its own local controller. This local controller is responsible for indexing and regulating accesses to its respective loop buffer. The main contributions in this chapter of the book are as follows:

- A distributed local controller based loop buffer organization, that can efficiently support two modes – single threaded and multi-threaded.
- In addition to executing loop nests sequentially and executing multiple compatible loops in parallel, the distributed controllers enable the execution of multiple *incompatible* loops in parallel inside one "virtual" thread.
- Simulation results show that the distributed controller based instruction memory hierarchy is energy efficient and scalable. Additionally, this enhancement improves the performance by enabling local data sharing instead of communication via the memories.

The rest of this chapter is organized as follows: Section 6.2 motivates the need for a multi-threaded architecture and the requirements for such an architecture. Section 6.3 presents the proposed architectural extension for performing multi-threading in an uni-threaded architecture and also illustrates how such an architecture can be used. Section 6.4 presents how a compilation technique can be built for this architecture. A qualitative comparison with related work in the area of multi-threading and distributed instruction memory is performed in Section 6.5. The quantitative results and comparisons are presented in Section 6.6 on various benchmarks. Finally, Section 6.7 concludes this chapter and summarizes the key messages that can be taken from this contribution.

6.2 Need for light weight multi-threading

The need for a light weight distributed multi-threading similar to other contributions of this book can be motivated across abstractions levels. Observations can be made from both from the application side as well as from the physical design perspective as motivated in Chapter 3.

Application perspective From the application side, the majority of the execution time in embedded applications is spent in loop code. Instead of

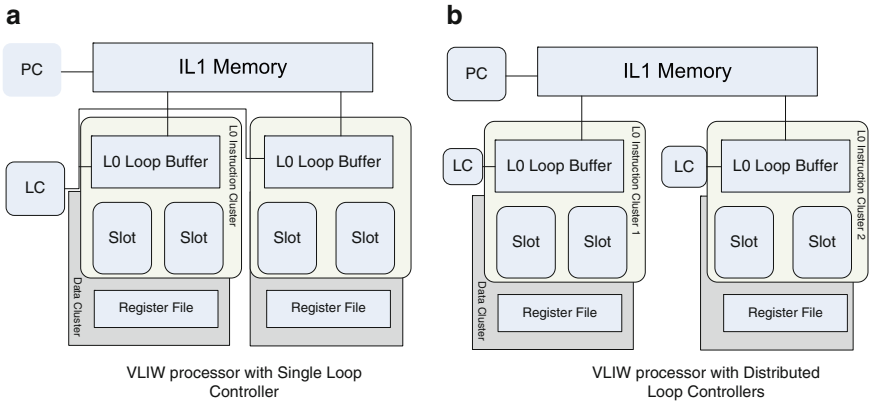


Figure 6.2: Processor architectures with central and distributed loop control

accessing the large L1 instruction memory for every instruction, the loop code can be buffered inside a smaller local memory called a loop buffer to reduce the instruction memory energy consumption. An architecture consisting of a conventional loop buffer based instruction memory is shown in Figure 6.2a. Another property of embedded systems is that the amount of coarse grained parallelism across applications is low as the number of threads running in parallel is low. Therefore the parallelism has to be exploited at the sub-task level, across different loops of the same application (which may have dependencies).

For executing loops, usually software instructions are used to keep track of the current iteration of the loop. This leads to instructions which decrement a register, compare and conditionally branch. Since looping is very common for embedded systems, it is beneficial to convert these instructions into a hardware based loop counter. This is now done in nearly all state-of-the-art DSPs generally known as zero overhead looping (ZOL). ZOL is a key feature in most DSP from the late 1980s [TI00, STM00, AT90, Eyr99]. These counters are referred to in the rest of the book as *iteration counters*. However these DSPs do not support execution of multiple incompatible loops in parallel (do not support SMT). Therefore room for improvement still exists with respect to exploiting parallelism across loops.

Physical design perspective From the layout perspective, it has been shown in [Dal05, Jos06, Syl99] and various other works that interconnect scaling is an key issue for energy-aware design. It is therefore crucial that the most frequently accessed instruction components for different clusters of the VLIW are located closer to their execution units. A distributed L0 buffer configuration for each VLIW cluster with separate loop controllers as shown

in Figure 6.2a and b, can significantly reduce the energy consumed in the most active local wiring. Further the loop buffers' width and depth can be customized for each cluster [Jay05a, Vda05].

Example illustration To illustrate the above requirement for a low power instruction memory hierarchy consider the following example: Different loops inside the same application can have very different loop characteristics (e.g memory operation dominated vs. computation dominated or address generation loop vs. data computation loop which may have different loop boundaries, loop iterator strides etc.). One such example code shown in Figure 6.1 illustrates two loops with different loop organizations. Code 1 gives a loop structure for the computational code that would be executed on the data path of the processor. Code 2 gives the loop structure for the corresponding code that is required for data and address management in the data memory hierarchy that would be executed on the address management/ generation unit of the processor. This may represent the code that fetches data from the external SDRAM and places it on the scratch-pad memory (or other memory transfer related operations). Code 1 in this example executes some operations on the data that was fetched by Code 2. In the context of embedded systems with software controlled data memory hierarchy, the above code structure is realistic.

The above code example can be mapped on different platforms. These two codes could also represent two parts/clusters of a VLIW executing two blocks of an algorithm, where each cluster could be customized for executing that particular block. In case these *loops were merged* on a platform with a single thread of control, extra conditions would need to be inserted. This would lead to poor performance, high energy consumption as well as an increased code size. Hence a need exists for a distributed control of two or more separate sets of codes.

The above code can also be run on two *separate processors*. In which case the each of the two processors would have separate PCs (therefore separate threads of control). However the two processors would not share data between their register files or between their L1 data memories. Explicit copy of data would be needed across the L1 data memories to share data. Therefore it is easy to imagine that the overhead of a fully multi-processor based solution would be very high for tightly coupled communicating loops.

In addition to the above example, one can also imagine these multiple threads as a chain of producers and consumers where the latency/ performance of these can be very different. This is often common in most code segments and therefore such a distributed multi-threading can benefit most applications.

From the above discussions of both application as well as layout, it can be summarized that the instruction memory for a low power embedded processor should satisfy the following requirements to be low power:

- Smaller memories (loop buffers) instead of large instruction memory.
- Specialized local controllers with minimal hardware overhead.
- Distributed and localized instruction memories to reduce long interconnect and minimized interconnect switching on very active connections.[1] Customized and distributed loop buffer for each cluster in terms of depth and width.
- Distributed local controllers that can support execution of different loop organizations in parallel (single loops, multiple compatible loops and multiple incompatible loops).

6.3 Proposed multi-threading architecture

This section presents the details of the proposed architectural enhancement that saves energy consumption and improves performance by enabling a synchronized multi-threaded operation in a uni-processor platform.

6.3.1 Extending a uni-processor for multi-threading

This work extends a uni-processor model to support two modes of loop buffer operation: Single-threaded and Multi-threaded. The former is the conventional one and the extension to the multi-threaded mode is achieved with special concern to support L0 buffer operation. A VLIW instruction is divided into bundles, where each bundle corresponds to an L0 cluster. To correctly control the execution of loops from a loop buffer, two types of counts come into play: a counter for indexing into loop buffer and an iterator counter. The indexing counter goes over all the instructions in the loop buffer (also referred to as LC), corresponding to a single iteration. The iterator counter keeps track of the current iteration number of the loop. The indexing counter is always implemented in hardware. However the iteration counting (or loop control) can be done in hardware or software. This book proposes two basic variants for of the architecture: a software counter based loop controller (shown in Figure 6.3) and a hardware counter based loop

[1] From simulations it appears the interconnect cost even in TSMC90nm technology is about 30% of the loop buffer memory access cost.

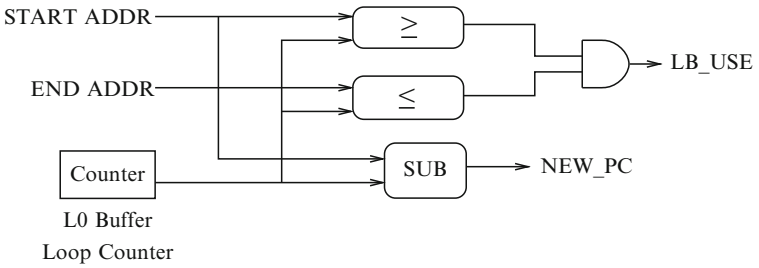


Figure 6.3: L0 Controller for every L0 instruction cluster based on software loops

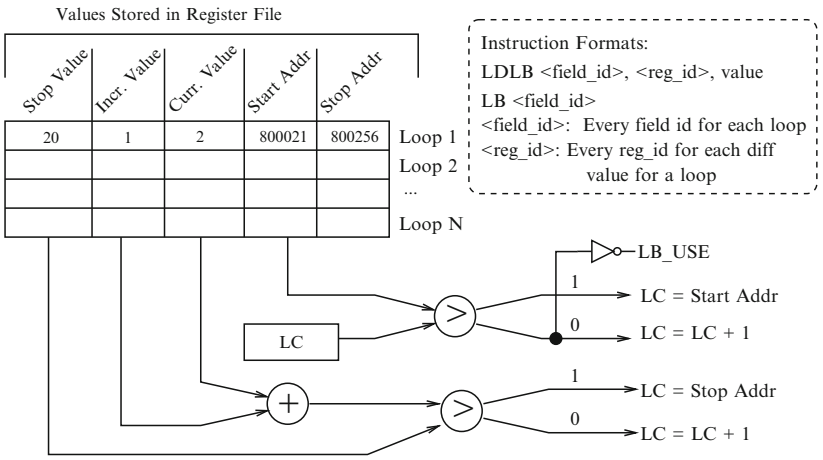


Figure 6.4: L0 Controller based on hardware loops

controller architecture (shown in Figure 6.4). The software counter based loop controller implements the iteration counter in software whereas the hardware loop counter based architecture implements the same in a separate hardware architecture.

Note that however fundamentally both software control based as well as hardware control based distributed loop controller architecture *can be used to accelerate non-loop code* as well. However given that in embedded systems, loop code is dominant the focus in this book has been limited to loop code alone.

6.3.1.1 Software counter based loop controller

An L0 controller (LC) (illustrated in Figure 6.3) along with a counter (size of counter depends on the depth of the loop buffer) is used to index and

regulate accesses to the L0 buffer. Unlike conventional Program Counters (PCs), the L0 controller (LC) logic is optimized to handle only loops, which leads to a smaller area and lower energy consumption. In other words, the PC can address complete address space of the instruction memory hierarchy, the L0 controller can access only the address space of the loop buffer. The *LB_USE* signal indicates execution of an instruction inside the L0 buffer. The *NEW_PC* signal is used to index into the L0 buffer. The loop buffer operation is initiated on encountering the LBON (Loop Buffer ON) instruction as mentioned in Chapter 5 and [Jay05b]. It is possible to perform branches inside the loop buffer as a path exists from the loop controller to the branch unit similar to the one presented in Chapter 5 and [Jay05b]. The L0 controller (LC) illustrated in Figure 6.3 only performs the counting to index into the loop buffer, therefore instructions to perform the operations on the loop iterator like increment, compare and conditional branch are still needed. This can be eliminated using a hardware based counter described in the next section.

6.3.1.2 Hardware counter based loop controller

Figure 6.4 shows an illustration of a hardware loop based architecture that performs the loop iterations. Note that this is still a fully programmable architecture. A register file contains the following: *start value, stop value, increment value of the iterator, start and stop address* for each of the different loops. Keeping these variables in the standard register file may be a problem as the register pressure is often too high given the large number of live variables. Therefore it is advisable to keep these variables can be kept in a separate register file. However they may also be kept in the standard register file as well, if the cost is not too high. In case these values are needed for computation, an explicit inter-cluster copy operation may be inserted. The current iterator value is also stored in a separate register as shown in Figure 6.4. Based on these values, every time the loop is executed, the corresponding checks are made and necessary logic is activated.

Figure 6.5 shows a sample C code and the corresponding assembly which is required for operating on this hardware based loop controller. At the beginning of the loop nest, the corresponding start, stop, increment values of the loop iterator and the start and stop address of the corresponding loop must be initialized. The LDLB instructions are used to load the start, stop, increment values of the iterators, and start, stop address of the loop respectively in the register file. The format for the LDLB instruction is shown in Figure 6.4. It can be seen from Figure 6.5b that although a number of LBLD instructions are needed to begin the loop mode (introducing an initial performance penalty), only one instruction (LB instruction) is needed while operating in the loop mode (LB 1 and LB 2). The loop buffer operation is started on encountering the LBON instruction, which demarcates the loop mode. The LB instructions activate the hardware shown in Figure 6.4 thereby performing the iterator

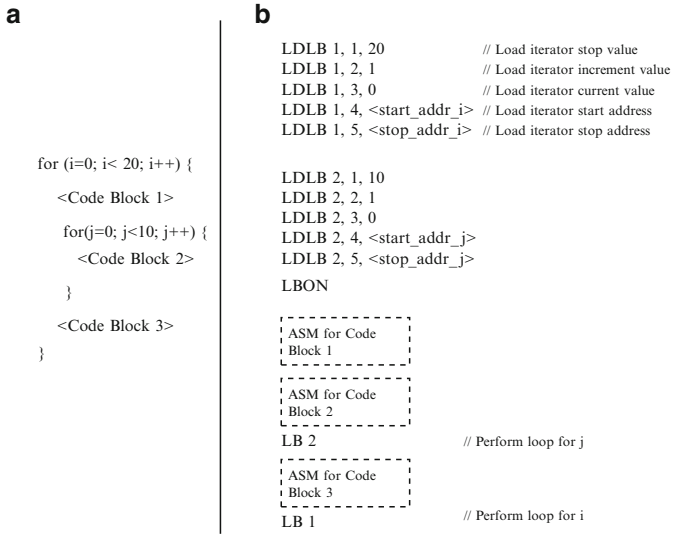


Figure 6.5: Assembly code for hardware loop counter based solution

increment/decrement, comparison operations for the loop and branching to the appropriate location if necessary. Hence the instruction memory cost (number of accesses to the loop buffer) for every loop is reduced, although the operations performed are the same. Such an architecture also allows the loop bounds to be both non-affine and non-manifest. Non-manifest implies that the initialization values are not known at compile time, non-affine means the operators are not linear. It is possible to have both conditions inside the loop buffer mode as well as breaks outside the loop buffer code.

Similar to the software based controller the signal *LB_USE* is generated for every loop to indicate that the loop buffer is in use. This signal is used later on for multi-threading (see Section 6.3.1.3).

Since a hardware iterator counter is used instead of using the regular datapath to perform the iterator manipulation, the counter size can be customized to be of the size of of the largest iterator value that may be used in the application which is much lower than the 32-bit integers.

6.3.1.3 Running multiple loops in parallel

The L0 controllers can be seamlessly operated in single/multi-threaded mode. The multi-threaded mode of operation for both the software controlled architecture and hardware controlled architecture is similar as both of them produce the same signals (*LB_USE*) and use LBON for starting the L0 operation. The state diagram of the L0 buffer operation is shown in Figure 6.6.

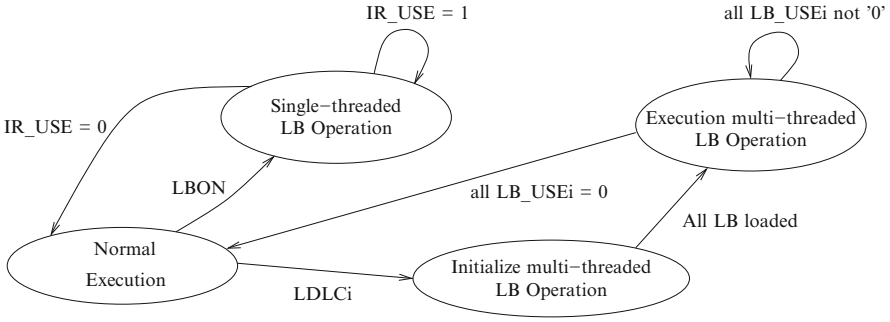


Figure 6.6: A state diagram illustrating the switching between single and multi-threaded mode of operation for both software controlled loop buffer and hardware controlled loop buffer

The single threaded loop buffer operation is initiated on encountering the *LBON* <*addr*> <*offset*> instruction. Here <*addr*> denotes the start address of the loop's first instruction and <*offset*> denotes the number of instructions to be fetched to the loop buffer starting from address <*addr*>. In the single threaded mode, the loop counter of each cluster is incremented in lock-step every cycle. This mode of operation is similar to the L0 buffer operation presented in [Jay05a, Jay05b], but in the proposed approach an entire cluster can be made inactive for a given loop nest to save energy. In case of the hardware based loop buffer operation the LDLB and LB instructions are also needed for the single threaded operation as explained in the previous section.

In the multi-threaded mode, the loop counters are still incremented in lock-step under the same clock, but not necessarily at every instruction. Instead alignment can be enforced at loop boundaries or explicit alignment points identified by the compiler (explained in Section 6.4). To spawn execution of multiple incompatible loops that have to be executed in parallel, each L0 cluster is provided with a separate instruction (*LDLCi* <*addr*> <*offset*>) to explicitly load different loops into the corresponding L0 clusters. Here *i* denotes the cluster number. For instance, in the following example two instructions LDLC1 <addr1> <offset1> and LDLC2 <addr2> <offset2> are inserted in the code to indicate that the loop at *addr1* is to be executed in cluster 1 and the loop at the *addr2* is to be executed in cluster 2.

```
            —
            LDLC1 <addr1> <offset1>
            LDLC2 <addr2> <offset2>
addr1:      for (...){
            Loop Body }
addr2:      for (...){
            Loop Body}
            —
```

Once the instruction *LDLCi* is encountered, the processor operates in the multi-threading mode. During the initialization phase all the active loop buffers are loaded with the code that they will be running. For example, the *i*th loop buffer will be loaded with *offseti* number of instructions starting from address *addri* specified in instruction *LDLCi*. Meanwhile, each cluster's loop controller copies the needed instructions from the instruction memory into the corresponding loop buffer. If not all the clusters are used for executing multiple loops, then explicit instructions are inserted by the compiler to disable them. The LDLCi instructions are used the same way and are used instead of the LBON instruction for both the software and hardware controlled loop buffer architectures. For the above example, in case of the hardware based loop buffer architecture, the LDLB instructions for initializing the loop iterations and address for the two loops would precede the LDLC instructions.

When a cluster has completed fetching a set of instructions from its corresponding address, the loop buffer enters the execution stage of the *Multi-threaded* execution operation. During the execution stage, each loop execution is independent of the others. This independent execution of the different clusters can be either controlled by the software or the hardware based loop controller mechanism. Although the loop iterators are not required to be in lock-step, the different loop buffers are aligned at specific alignment points (where dependencies need to be met) that are identified by the compiler. Additionally, the compiler or the programmer must ensure the data consistency or the necessary data transfers across the data clusters. programmer or the compiler.

The loops loaded onto the different L0 buffers can have different loop boundaries, loop iterators, loop increments etc. This enables executing different incompatible loops in parallel.

6.4 Compilation support potential

The code generation for the proposed architecture is similar to the code generated for a conventional VLIW processor, except for the parts of the code that need to be executed in multi-threaded mode. As mentioned in the previous section, additional instructions are inserted to initiate the multi-threaded mode of operation and its associated communication and synchronization operations.

However in this work the compiler has not been automated. For the experiments that follow, insertion of the instructions/operations have been done manually. A more detailed study is required for an efficient compilation step for such a multi-threaded architecture. This section describes a possible road

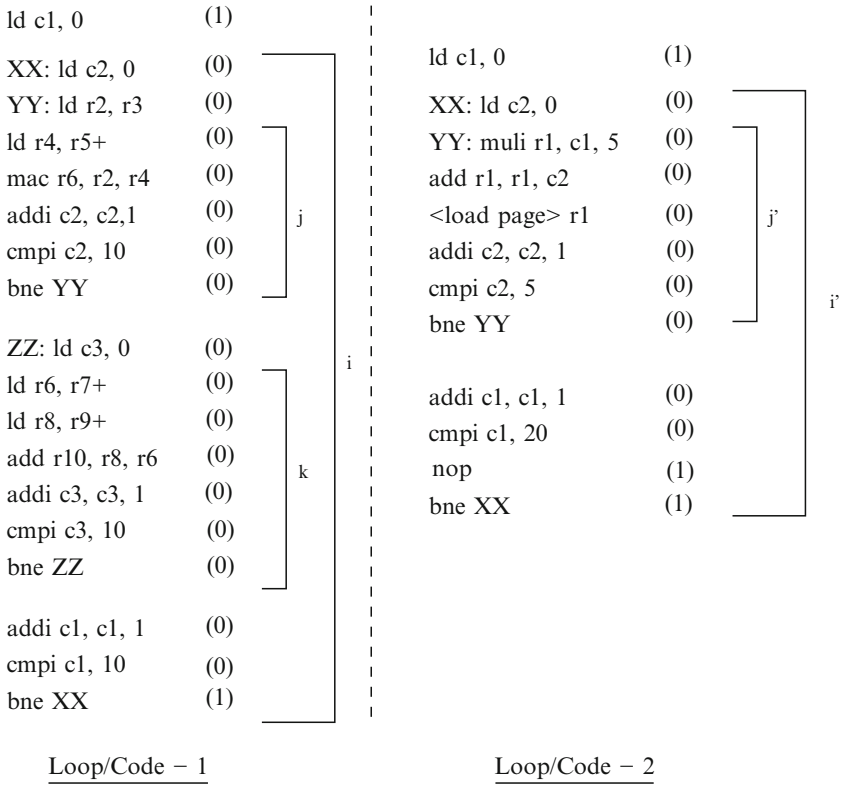


Figure 6.7: Assembly code for the code shown in Figure 6.1 (extra alignment bits are shown in brackets)

to building a compiler for such a multi-threaded architecture and the work on the compiler is to be considered as possible future directions of this research.

Figure 6.7 shows the assembly code for the two incompatible loops presented in Figure 6.1. Code 1 is loaded to L0 Buffer 1 and Code 2 is loaded to L0 Buffer 2. As for two iterations of loop *i*, only one iteration of loop *i'* has to be executed, there is a need to identify this dependency and insert the necessary alignment points to respect this dependency. For this purpose, the two loops shown in Figure 6.1 are first represented in a polyhedral model [Qui00]. Once the different codes are represented in a common iteration domain [Qui00], a data dependency analysis can be done [Gom04]. On analyzing the data dependencies between different codes, the alignment points can be derived. The alignment points are then annotated back on the original code shown in Figure 6.7 within brackets. In case the original code has pointers or if conditions which prevent from entering the polyhedral model, various pre-processing techniques like SSA, if-conversion, pointer removal

[Pal05] are used. These preprocessing steps are typically executed on top of a polyhedral framework and they fall outside the scope of this book. Once the different code segments are represented in the common iteration domain, various algorithms like Kernigham-Lin etc. can be used for a min-cut through the polyhedra to ensure low communication across the two threads. Moreover at the cuts, appropriate cluster copy/communication instructions can be inserted if and when required. Further, such a cut across would also need to ensure that the ILP and DLP is balanced on the different threads.

Once the alignment has been identified either by the compiler or manually, it can be implemented between the two clusters by adding an extra bit to every instruction, as shown in Figure 6.7. A '0' bit indicates that the instruction can be executed independently of the other cluster. A '1' bit indicates that the instruction can only be executed if the other cluster issues a '1' as well. In case the cluster 1 issuing a '1' bit, does not get a '1' from the other cluster, then cluster 1 would stall till it gets the alignment bit from the other cluster.

For the example shown in Figure 6.7, only one extra bit is sufficient, as only two LCs exists. In case of more than two LCs, one bit can be used for every other cluster that needs to be synchronized with. At the worst case, this would require as many bits as one less than the number of clusters. A trade off can be made between granularity of synchronization versus the overhead due to synchronization. This instruction level alignment reduces the number of accesses to the instruction memory and hence improves the energy-efficiency.

It can be seen from the assembly code in Figure 6.7, that using the alignment bits the data sharing can be done at the register level. This is in contrast to the cache level like used in the case of SMT processors. This reduces the number of reads and writes to the memory and register file and further improves performance and energy usage.

6.5 Comparison to related work

The related work to the proposed distributed loop buffer based architecture can be categorized into various parts: (1) loop buffers or L0 organization, (2) loop transformation techniques, (3) Simultaneous multi-threading techniques, and (4) multi-core.

loop buffers/L0 organization The *L0 organization* is a commonly used technique to reduce instruction memory hierarchy energy [Cot02, Jay05b]. In this architecture, a small loop buffer is used in addition to the large

instruction caches/memories, which is used to store only loops or parts of loops. Additionally, several compiler techniques are proposed to improve energy and performance of loop buffering [Sia01, Ste02]. State-of-the-art instruction memory organization can be categorized based on various aspects like loop buffers, local controllers, thread of controls etc. This work completely reuses the space of loop buffers and various scheduling techniques for loop buffers [Cot02, Jay05a, Jay05b, Kob07b].

Most state-of-the-art loop buffers and the associated local controllers [Cot02] are centralized. However for higher energy efficiency both the loop buffers and local controllers can be distributed.

Additionally, the thread of control can be single or multiple threaded. State of the art loop buffer organizations are intended for single thread of control (as illustrated in Figure 6.2a). This book proposes the support for the execution of multiple threads, in particular for the execution of multiple incompatible loops in parallel. In order to support this multiple loop execution, the local controllers need to have additional functionality as detailed in Section 6.3. Local controllers in [Jay05b] only regulate the accesses to the loop buffers. In contrast, local controllers in the proposed approach provide indexes to the loop buffers and synchronize with other local controllers, in addition to regulating the access to the loop buffers. Some commercial processors like [Sta00] implement the unified loop controller as a hardware counter, but enforce restrictions on handling branches during the loop mode. Other limitations include, the need for affine loop bounds for performing loop iterations in hardware. In the proposed method, branches can be present inside the loop mode, either as a branch inside the loop buffer or as a branch outside the loop buffer contents (as explained in Section 6.3).

Loop transformations In processors with a single thread of control (Figure 6.2a) *loop transformations* are often used like *loop fusion* is a commonly used technique to execute multiple threads in parallel. By applying loop fusion, the candidate loops with different threads of control are merged into a single loop, with a single thread of control thereby converting thread-level-parallelism (TLP) into instruction-level-parallelism (ILP). However, with this technique incompatible loops like the one shown in Figure 6.1 cannot be handled efficiently. When incompatible loops are merged, many *if-then-else* constructs and other control statements are required for the checks on loop iterators. The number of these additional constructs needed can be very large, resulting in loss of both energy and performance (see Section 6.6 for comparison). This overhead still remains even if advanced loop morphing [Gom04, Sca06] technique is applied.

Multi-threaded architectures Multi-threaded architectures and Simultaneous Multi-Threaded (SMT) processors [Oze05, Kax01, Tul95] can also

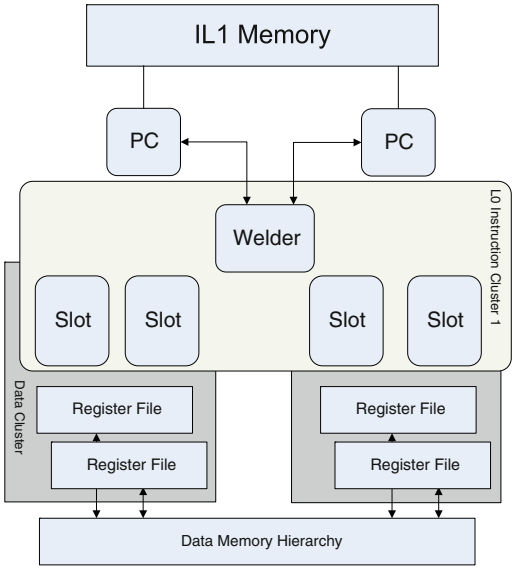


Figure 6.8: Weld based SMT architecture as in [Oze05]

execute multiple loops in parallel. In such architectures, each thread has a set of exclusive resources to hold the state of the thread. One such SMT based technique where each thread has its own register file and program counter logic, is shown in Figure 6.8. Furthermore, in these architectures the data communication between the processes/threads is done at the cache level (or level-1 data memory). While fundamentally both techniques can be used for both loop and non-loop code, the proposed multi-threading architecture has been developed keeping in mind the need to accelerate tightly coupled loops. Compared to SMT architectures, the proposed multi-threaded architecture has the following differentiators. Firstly, the hardware overhead/duplication is minimal. A simplified local controller is provided for each thread. Secondly, the data communication between the threads, in addition to cache level (or level-1 data memory) can also be done at the register file level. Thirdly, the proposed architecture is intended specifically for executing multiple incompatible loops. This implies, that any generic threads cannot be executed in the architecture unless the generic threads are pre-transformed into loops. Since the hardware overhead is minimal, the proposed architecture is energy efficient. This is quantitatively shown in Section 6.6).

From the software perspective, the data and control dependencies between the two threads can be analyzed through design/compile time analysis of the loops as described in Section 6.4. This analysis improves the performance

and energy efficiency, as it enables efficient data communication between the threads through the register file level and insertion of synchronization points between the loops. In Multi-threaded or SMT processors, such analysis is not performed. The primary motivation for SMT processors is to increase performance/throughput/IPC of the processor. In other words, this improves the net-throughput of the processor and not the single-thread performance. This is achieved by improving resource utilization, i.e., fill in the empty instruction cycles of functional units (FUs) with instructions from different threads without view of the complete code (limited scope). Hence, all the threads share all the FUs in the datapath. In the proposed architecture, the primary motivation is to improve resource utilization and by doing this both primarily reduce *energy* consumption and improve performance. As motivated in the earlier sections, each thread has an exclusive set of FUs (FUs in one cluster) to minimize interconnect energy and the loops are pre-processed such that computations in each thread use only their exclusive set of FUs. This pre-processing consists of the mapping of loops to individual clusters.

Multi-core A link can also be made between the proposed multi-threading architecture and *multi-core systems*. For a multi-core platform, the communication costs between cores should be minimized by placing applications with minimal communication on different cores. The proposed technique applies to parallelism present inside the same application where there is communication between the loops. Therefore the proposed technique would be complementary to a multi-core platform, where parallelism present due to multiple applications is exploited across cores. And the parallelism present inside a single application can be still exploited inside one core using the technique presented in this chapter.

With the proposed architecture enhancement, multiple incompatible loops can be executed in parallel, without the overhead/limitations mentioned above. Firstly, multiple synchronizable Loop Controllers (LCs) (one LC per loop) enable the execution of multiple loops in parallel. Secondly, such a distributed technique enables a reduction in the interconnect required between the instruction memory and the datapath. Thirdly, the LC logic is simplified compared to a full fledged program counter (PC) and the hardware overhead is minimal, as it has to execute only loop code. Fourthly, data sharing and synchronization is done at the register file level and therefore context switching and management costs are eliminated. This contribution of the book also proposes a hardware based loop counter which is capable of having breaks out of the loop (instruction affects the PC) and conditional/unconditional jumps inside as well (instruction affects the LC and counters). Non-affine and non-manifest loop bounds (which occur when the loop bounds are only known at run-time) can also be supported.

6.6 Experimental results

Section 6.6.1 presents the experimental setup that is used to demonstrate this work and Section 6.6.2 describe the benchmarks that have and the baseline architectures that have been used in the experiment. Section 6.6.3 analyzes the energy and performance gains of the proposed architecture and the reasons for the gains.

6.6.1 Experimental platform setup

The experiments are performed on the COFFEE framework as described in Chapter 4. The target technology used is *TSMC90nm G* technology, 1.0 V Vdd. The complete system is clocked at 200 MHz,[2] which is a realistic frequency for an embedded processor. The power/energy model for each of the individual components have been obtained after place and route and extracted gate level simulation.

Given that the proposed architecture has been optimized while taking into account a distributed organization it would scale well with future technologies as well as other operating frequencies used in embedded processors. The extra energy consumed due to the synchronization hardware is also estimated after physical design after layout, capacitance extraction and back-annotation. The Artisan Memory Generator [ARM] has been used for generating the memories. These different blocks haven then been placed and routed, and the energy consumption of the interconnect between the different components is calculated based on the activation of the different components.

The interconnect requirement between the loop buffers, loop controller and the functional units has also been taken into account while computing the energy estimates.

Special instructions as mentioned in the previous sections have been inserted to enable multi-threaded operation on the VLIW. Since most current embedded applications do not provide very high ILP, a VLIW of four slots was chosen. All slots are homogeneous and form one data cluster i.e. all four slots share the same global register file and two L0 instruction clusters of two slots each. Although the proposed multi-threading technique has been applied on a 4-issue VLIW, the results scale to other sizes of VLIWs provided the application also provides the required ILP. In case more threads are used (greater than 2), a wider VLIW can be used (based on the ILP available and the per-

[2] 200 MHz can be considered roughly the clock frequency of most embedded systems. The conclusions would still be valid for other operating frequencies.

formance requirement). The input on the number of slots for each cluster and its functionality is one of the crucial inputs for the compiler to map the appropriate code on either side of the cluster.

6.6.2 Benchmarks and base architectures used

The TI DSP benchmarks [TI09d] have been used for benchmarking the proposed multi-threading architecture, which is a representative set for the embedded systems domain. The output of the first benchmark is assumed to be the input to the second benchmark. This is done to create an artificial dependency between the two threads. Experiments are also performed on real kernels from a Software Defined Radio (SDR) design of a MIMO WLAN receiver (two-antenna OFDM based outputs) [Pal08]. After profiling, the blocks that contribute most to the overall computational requirement were taken (viz. Channel Estimation kernels, Channel Compensation). In these cases, dependencies exist across different blocks and they can be executed in two clusters.

Figures 6.9 and 6.10 respectively, show the energy savings and performance gains that can be obtained when multiple kernels are run on different L0 instruction clusters of the VLIW processor with the proposed multi-threading extension.

In the *Sequential case* (Baseline case), two different codes are executed on the VLIW one after the other. The VLIW has a centralized loop buffer organization. In the loop merged case, a variant of the loop fusion technique proposed

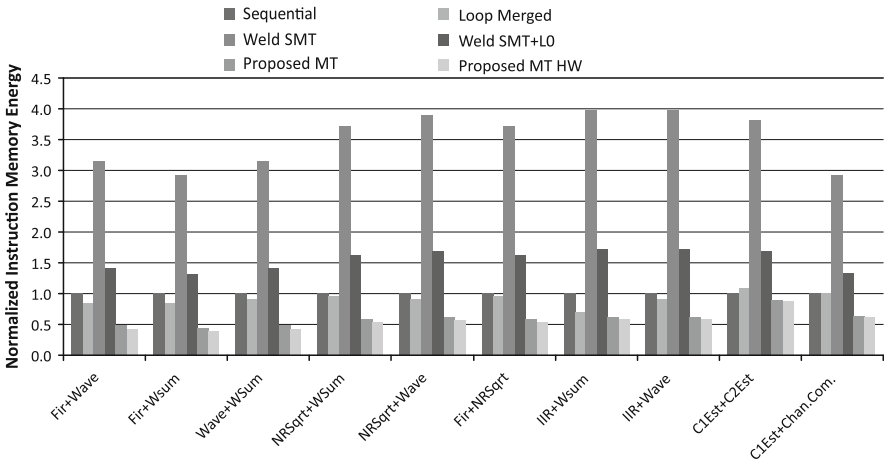


Figure 6.9: Instruction memory energy savings normalized to Sequential Execution

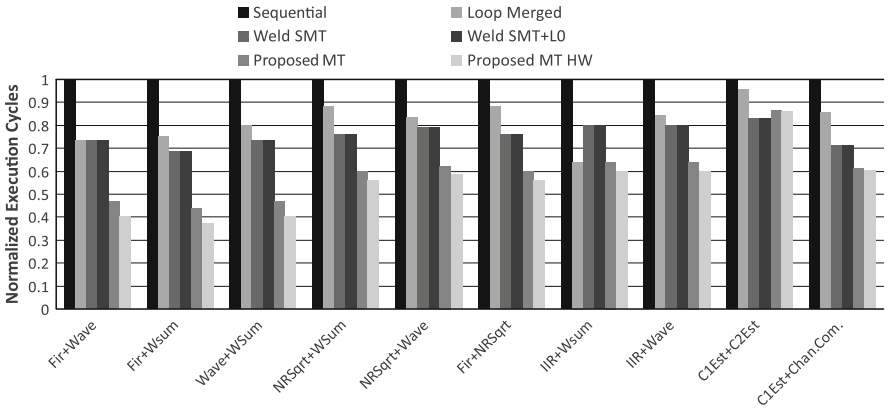


Figure 6.10: Performance comparison normalized to sequential execution

in [Gom04] is applied and executed on the VLIW with a centralized loop buffer organization and with a central loop controller. For the *Weld SMT* case, a complete program counter and instruction memory of 32KB are used. The SMT is performed as described in [Oze05]. This SMT has also been enhanced with an energy efficient centralized loop buffer instead of the IL1 and PC based architecture. The overhead of the "Welder" is also taken into account. The "Welder" is a network constructed of muxes and a mux controller, to distribute operations for different threads over the functional units. Although the SMT and Loop buffer technique are orthogonal, for the comparison to be fair, loop buffering technique is also applied to the SMT architecture (*Weld SMT+L0*).

The software based proposed multi-threading (*Proposed MT*) is based on the logic shown in Figure 6.3. The hardware loop counter based multi-threading (*Proposed MT HW*) is based on the logic shown in Figure 6.4. This architecture has a 5-bit iterator counter logic for each cluster. All the results are normalized with respect to the sequential execution. Also aggressive compiler optimizations like software pipelining, loop unrolling etc. have been applied in all the different cases.

6.6.3 Energy and performance analysis

The *Loop-Merged* (Morphed) technique saves both performance and energy over the *Sequential* technique (see Figures 6.9 and 6.10) since extra memory accesses are not required and data sharing is performed at the register file level. Therefore the *Loop-Merged* technique is more energy as well as performance efficient compared to the *Sequential* case. In case of the *Loop-Merged* case an overhead exists due to iterator boundaries etc., which introduce extra

control instructions. Furthermore loop merged case would also give an extra overhead in terms of code size required.

The *Weld SMT* and *Weld SMT+L0* improve the performance further as both tasks are performed simultaneously. In some benchmarks used, the *Weld SMT* can help achieve an IPC which is close to 4. The overhead due to the "Welder" is quite large and hence in terms of energy the Weld based techniques perform worse than both the sequential and the loop merged case. Since the "Welder" has to be activated at every issue cycle, its activity is quite high. Additionally, an extra overhead is present for maintaining two PCs (in case of *Weld SMT*) or two LCs (in case of *Weld SMT+L0*) for running two threads in parallel. The data sharing is at the level of the DL1, therefore an added communication overhead exists. As a result, the *Weld* based techniques perform worse than the sequential and the loop merged techniques in terms of energy. Even if enhancements like sharing data at the register file level are introduced, the overhead due to the Weld logic and maintenance of two PCs is large for embedded systems. For an architecture which can run more than two threads, this overhead would be higher, whereas the proposed L0 controller is more scalable.

In case of the *Proposed MT* and *Proposed MT HW* architectures, the tasks are performed simultaneously (like in the case of Weld SMT), but the data sharing is at the register-level. This explains the energy and performance gains over the *Sequential* and *Loop Merged* cases. Since the overhead of the "Welder" is not present, the energy gains over the Weld SMT+L0 technique are large as well. Further gains are obtained due to the reduced logic requirement for the loop controllers and the distributed loop buffers. In conclusion, the proposed technique has the advantages of both loop-merging as well as SMT and avoids the pit-falls of both these techniques.

The results show that the *Proposed MT* has an energy saving of 40% over sequential, 34% over advanced loop merged and 59% over the enhanced SMT (Weld SMT+L0) technique. On average the *Proposed MT* has a performance gain of 40% over sequential, 27% over loop merged and 22% over Weld SMT techniques. In certain cases like *Chan1Est + Chan2Est* and *C1Est + ChanCompen*, the SMT based techniques outperform the proposed multi-threading as the amount of data sharing is very low compared to the size of the benchmark. In terms of energy consumption the proposed multi-threading is always better than other techniques. It can be intuitively seen that in case the *Weld SMT+L0* architecture is further enhanced with data sharing at the register file level, the *Proposed MT* and *Proposed MT HW* would perform relatively worse in terms of performance. In terms of energy efficiency however, the *Proposed MT* and *Proposed MT HW* based architectures would still be much better. It has been theoretically[3] observed that even when the *Weld*

[3]This implies removing the cycles that correspond to the shared data transfer through the memory.

SMT+L0 architecture would support data sharing at the register file level, the performance gain of this architecture over the *Proposed MT* and *Proposed MT HW* is less than 5% in most cases.

The *Proposed MT HW* is both more energy efficient as well as has better performance compared to the *Proposed MT* technique. This is more apparent in smaller benchmarks as the number of instructions per loop iteration is small. The hardware based loop counter (*Proposed MT HW*) outperforms the software based technique, as the number of cycles required for performing the loop branches and iterator computation is reduced. This difference is larger in case of smaller benchmarks and smaller in case of larger benchmarks. Also in terms of energy efficiency the *Proposed MT HW* is more energy efficient compared to the *Proposed MT*. The overhead of loading the loop iterators and the values required form the *Proposed MT HW* architecture was about two to three cycles for every loop nest. This overhead depends on the depth of the loop nest. Since all the LDLB instructions are independent of each other, they can be executed in parallel. Since in almost all cases, the cycles required for the loop body multiplied by the loop iterations is quite large, the extra overhead of initialization of the hardware counter is small. The synchronization required between the distributed loop buffers in case of both the *Proposed MT* and *Proposed MT HW*, was of the order of one or two cycles per loop iteration for most benchmarks. The relative overhead of this synchronization depends on the number of cycles required for the loop body itself and the amount of data sharing present across the two loops running in parallel. For example, the loop body size of the benchmark *Chan1Est + Chan2Est* is about 163 cycles and 6 cycles of this were due to synchronization.

To further analyze the energy efficiency of these various architectures, the energy consumption in different parts of the instruction memory is investigated for three of the benchmarks and is shown in Figure 6.11. The energy consumption is split into three parts and is normalized to the *Weld SMT + L0* energy consumption:

1. LB Energy: Energy consumption of the loop buffer which stores the loop instructions
2. LC Energy: Energy consumption of the control logic required for accessing the instruction (Loop Controller, Weld logic, Hardware loop counter etc.)
3. Interconnect (Inter.) Energy: Energy consumption of the interconnect between the loop buffer and the FUs

Figure 6.11 shows that the energy consumption of the LC logic considerably reduced as we move from the *Weld SMT+L0* based architecture to a standard *L0* based architecture with a single LC or the *Proposed MT* and *Proposed MT*

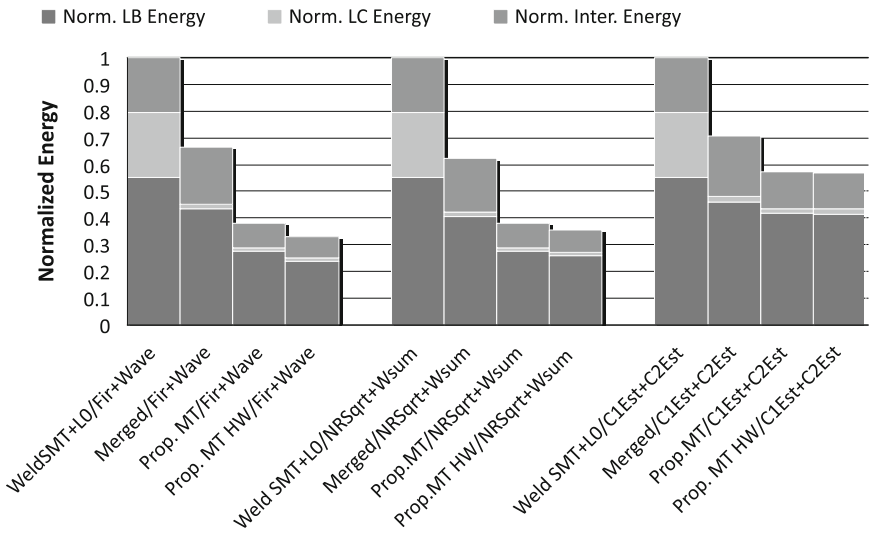


Figure 6.11: Energy breakdown of different architectures

HW based architectures. This is because of the overhead of the Weld logic and the extra cost of maintaining two loop controllers. The interconnect cost also reduces as we go from a centralized loop buffer based architecture to a distributed loop buffer based architecture by almost a factor of 20%. In case of smaller loops the energy efficiency of the *Proposed MT HW* is higher than that of the *Proposed MT*.

6.7 Conclusion and key messages of this chapter

This chapter has introduced one of the core contributions of this book namely a distributed loop buffer organization. The proposed architecture has a distributed control which allows the execution of multiple incompatible loops in parallel. Such an architecture has also been shown to provide efficient local communication while enabling running multiple threads in parallel. It has also been shown to be energy efficient compared to state-of-the-art SMT techniques as well as other techniques like loop merging. A high-level proposal for building a compiler technique supporting this architecture extension has also been included, to illustrate the compilability of the architecture.

CHAPTER 7

Handling Irregular Indexed Arrays and Dynamically Accessed Data on Scratchpad Memory Organisations

Abstract

Many application codes contain indirectly indexed arrays. These are arrays accessed inside loop-nests using linear function of iterators, symbolic constants and, additionally, other arrays. Several techniques are available in the literature to reorder the data accesses to such arrays in the given program in order to improve locality. However, we then still need a technique for mapping the data to the memory hierarchy, if the hierarchy consists of not just cache but scratchpad memory (SPM) as well. For the cache, data management is done automatically by instituting the $least\ recently\ used$ *algorithm in hardware. From the software perspective, all that is necessary is to present a program where much of the reuse has been well exposed. But for effective execution with SPM, a lot more than that is warranted, and for indirectly indexed arrays the problem is even further compounded. That problem is the first major focus of this chapter.*

A further complication is introduced when the data structures become dynamically accessed. In the existing literature, that now leads to a pure dynamic allocation process with new *and* $delete$ *functions that are treated by a dynamic memory manager embedded in the middleware. That potentially leads however to significant performance and energy overhead. Hence, whenever feasible such dynamic accesses should still be handled by a compile-time mapping technique, and preferably they should be allocated on a scratchpad again. That problem is the second major focus of this chapter.*

F. Catthoor et al., *Ultra-Low Energy Domain-Specific Instruction-Set Processors*,
DOI 10.1007/978-90-481-9528-2_7,

7.1 Introduction

Chapters 5 and 6 have extensively dealt with the instruction memory organisation. However, the data memory hierarchy remains a large bottleneck in terms of energy consumption as shown in Chapter 3. Given this bottleneck, in the embedded systems community, the concept of scratchpad memory (SPM) has lately gained significant prominence [Mar03, Pan98]. SPM is essentially comprised of an on-chip SRAM that occupies a small part of the address-space of the main memory. It is also considered as a software-controlled cache [Chi00, Zuc98], when the hardware is actually a cache which provides the feature that a part of its memory can be controlled through software. The advantage of the SPM is that because of its hardware simplicity it is smaller in size and consumes less energy on a per access basis [Ban02].

Most work in the past on scratchpad management has happened on arrays that are indexed using linear functions of the loop-iterators, i.e. directly indexed arrays [Ste02, Iss04, Kan04b, Ver07]. Also our previous work on the DTSE methodology (see [Cat98b, Cat02]) has provided several source code transformation steps which improve the usage of this SPM.

Many multimedia codes, however, also index into arrays using other arrays, i.e. incorporating indirectly indexed arrays. Current techniques map these indirectly indexed arrays either, as a whole [Pan97] onto the SPM, or reject it altogether if it does not fit completely in the SPM. Not incorporating the ability to analyze and move the relevant parts of the indirectly indexed arrays onto the scratchpad implies a potentially heavily suboptimal solution. Hence, we wish to address this energy bottleneck in this chapter.

The rest of this chapter is organized as follows. In Section 7.2 a real-life example is presented to illustrate the problem of mapping of indirectly indexed arrays to SPM. Section 7.3 reviews related literature. Section 7.4 describes concept of irregularity over an iterator, which together with the SPM cost model (Section 7.5) lead to our SPM mapping algorithm (Section 7.6). Section 7.7 presents the results on the indirect indexing. Section 7.9 presents related work on the dynamic data structure access. Section 7.10 discusses locality optimization and mapping of data structures with static data organization but with dynamic referencing behavior. We will look at the interesting example of the spell-checker implemented using the trie data structure. We will also prove using our Independent Reference Model that this class of application, under optimal SPM mapping, will always outperform the direct mapped cache. Section 7.11 next takes up the problem of locality optimization and mapping of data structures with dynamic data organization. We start with applications where the data organization can still be contained so that good mapping to SPM is still possible.

More precisely, we will look at the interesting example of Prim's algorithm for minimum spanning tree. We will then discuss much more dynamic cases.

Based on these techniques, we can largely reuse the highly efficient SPM mapping techniques that have been proposed in the past for regularly and directly accessed arrays, such as the DTSE methodology of [Cat98b, Cat02] and the techniques of [Ste02, Iss04, Kan04b, Ver07].

7.2 Motivating example for irregular indexing

The code segment below is from QSDPCM [Sto98], an inter-frame image compression algorithm. The arrays x and y are motion vectors which are used to index into the array $image$. Figure 7.1 (left side) is an illustration of this selection process. Now, while x and y are directly indexed, $image$ is indirectly indexed using iterators as well as the arrays x and y.

```
Example 1
     for( i = 0 ; i < N ; i++)for( j  = 0; j < M ; j++ )
       for( k = 0 ; k < 16 ; k++ ) for( l = 0 ; l < 16 ; l++ )
         ...= image[ 16*i+x[i][j]+k ][ 16*j+y[i][j]+l ];
```

Let us evaluate our options such that all references in the loop-nest map to the SPM. Arrays x and y (motion-vectors) are small in size, so they can be copied completely to the SPM before the start of the loop-nest. Also, since x and y are directly indexed, we know precisely the elements that would be accessed at a given iteration point. For instance, using data-tiling [Kan04b], we could move only a single row of x and y on SPM. The code for transferring

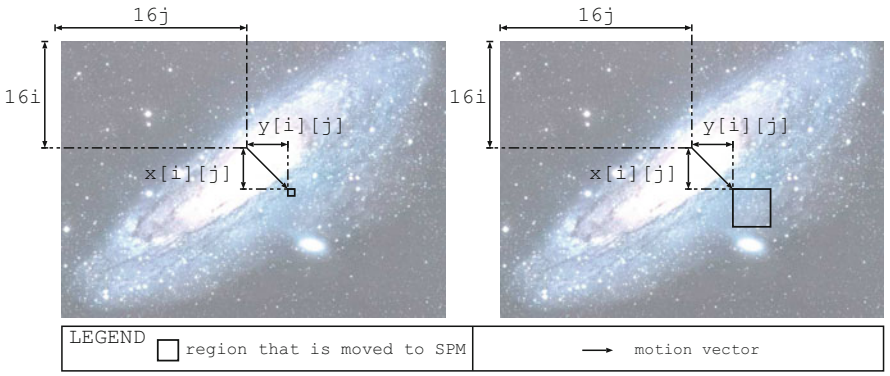


Figure 7.1: Migrating the relevant portion of an indirectly indexed image-data to SPM requires analysis of the index expression and indexing arrays

the relevant rows of x and y to SPM is shown below. We assume that a DMA controller is available to move data efficiently to SPM (such as the one described in [Pol04]).

```
for( i = 0 ; i < N ; i++){
  SPM[0:M-1]    <- x[i][0:M-1] //DMA-transfer
  SPM[M:2*M-1] <- y[i][0:M-1] //DMA-transfer
  for( j  = 0; j < M ; j++ )
    for( k = 0 ; k < 16 ; k++ )
      for( l = 0 ; l < 16 ; l++ )
        ...= image[ 16*i+SPM[j]+k ][ 16*j+SPM[M+j]+l ];}
```

Next, consider the mapping of $image$ to the SPM. Since it is indirectly indexed, current techniques would try to map it as a whole. But that may not be always possible due to limited SPM size (e.g. a 256×256 image would need 64 KB of SPM). The problem with trying to map indirectly indexed arrays is that because of the indirection we may not know easily, at compile-time, which locations of the array would be accessed. This situation is unlike those for regular computations such as matrix multiplication.

However, in this chapter we will show that for some cases of indirectly indexed arrays, it is still possible to do a good SPM mapping. For example, when we analyze the index expression of $image$ we see that for a fixed value of i and j the set of elements of $image$ that are accessed (inside loop of k and l) is completely known. These locations are, of course, relative to the starting point defined by $x[i][j]$ and $y[i][j]$, which are known only at run-time. One possible mapping (data-tiling) of image is shown below. Each time, for a given value of i and j, the 16×16 block in $image$ that would be next accessed is copied to SPM. This process is also shown in Figure 7.1 (right side).

```
          for( i = 0 ; i < N ; i++)
            for( j  = 0; j < M ; j++ ){
SPM[0:15][0:15]<-image[16*i+x[i][j]+(0:15)]
         [16*j+y[i][j]+(0:15)];
              for( k = 0 ; k < 16 ; k++ )
                for( l = 0 ; l < 16 ; l++ )
                  ...= SPM[k][l];     }
```

7.3 Related work on irregular indexed array handling

Several researchers have presented mapping techniques for the SPM [Domi05, Kan01b, Ver04]. Initial work, by Panda et. al. [Pan97], mapped only small-sized arrays on SPM based on frequency of access.

Their approach was extended and improved upon by other researchers [Avi02, Ste02, Ver04, Ver07] who applied knapsack formulation and ILP solvers to find the best set of data objects (globals, stack variables and arrays) and program routines that would still fit into the SPM and yet save the highest amount of energy. For arrays larger than the SPM size, these solutions do not work well. However, for arrays accessed in a regular pattern – inside nested-loops by index expressions which are linear functions of the loop iterators – several additional SPM mapping techniques using data-space and iteration space tiling have been proposed [Iss04, Kan01a, Kan04b]. They work well irrespective of the array size, and always outperform the cache.

Kandemir et al. [Kan04b] consider both loop and data-layout transformations to arrive at the best SPM mapping for directly indexed arrays. Issenin et. al. [Iss04] consider multi-level SPM and use it to exploit multi-level locality. The locality optimizations are assumed to have been done prior to their step. However, all these works consider only directly indexed arrays.

In [Abs05], preliminary ideas are presented on mapping of indirectly indexed arrays to SPM. The focus is on deciding which portion of the array to move to SPM. The locality optimization issue has been addressed in [Abs06]. Subsequent to our work, other authors have taken indirectly indexed array analysis further, for instance, to provide algorithms for loop tiling of such references [Che06].

7.4 Regular and irregular arrays

Here, we describe the concept of regularity and irregularity of an array over an iterator. We will use this later to optimize the mapping to scratchpad.

First some notations will be introduced. A reference to an array inside a given loop-nest can be represented by a reference matrix and an offset matrix. For instance, `B[i+j]` can be represented as:

$$R_B\mathbf{I} + \mathbf{o}_B = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix}$$

Irregular: An m-dimensional array X, with a generic reference $\Re_X(\mathbf{I})$ is described below:

$$\Re_X(\mathbf{I}) = \begin{bmatrix} \sum_q U_{1,q}\langle\Re_{U_{1,q}}(\mathbf{I})\rangle \\ \sum_q U_{2,q}\langle\Re_{U_{2,q}}(\mathbf{I})\rangle \\ \vdots \\ \sum_q U_{m,q}\langle\Re_{U_{m,q}}(\mathbf{I})\rangle \end{bmatrix} + R_X\mathbf{I} + \mathbf{o}_X \tag{7.1}$$

In this equation, the m dimensions each have an expression that is potentially irregular, where several iterators from the loop nest iterator vector I are combined. This is represented by the Σ_q summation augmented with the $U_{k,q}$ selection. The generic reference $R_X I$ provides the linear baseline expression and o_X provides the constant offsets as indicated above.

This Eq. 7.1 is said to be "$irregular$ in the pth dimension, over iterator i", if at least one of the two hold:

1. For some q, i appears in the index expression of $U_{p,q}$.
2. For some q, $U_{p,q}$ is irregular over i in at least one of its dimensions. $\diamond$

For example, array C with reference $C[k][k+1]$, is regular over $\{k\}$ in both its dimension. Array B in $B\left[C[k][k+1] + D[j] + i\right]$, is irregular over $\{j,k\}$, but is regular over $\{i\}$. Array Y in $Y\left[Z[i][j] + j\right][k]$ is irregular over $\{i,j\}$, and regular over $\{j\}$, in the first dimension. It is regular over $\{k\}$ in the second dimension.

7.5 Cost model for data transfer

Consider a loop-nest with an iterator vector $\mathbf{I} = \left[\begin{array}{cccc} i_1 & i_2 & \cdots & i_n \end{array}\right]'$ where each level in the loop nest has its own iterator i_k. Suppose the iterator i_k assumes the values $1, 2, \ldots, I_k$. An array X referenced inside the body of such a loop-nest will, therefore, be accessed $I_1 \times I_2 \times \cdots \times I_n$ times. We have two scenarios:

1. Array X not mapped to SPM: In this case, every access to X is an access directly to the external-memory. If m_E is the cost per access for a word in the external memory, then the $Total\_Cost = I_1 \times I_2 \times \cdots \times I_n \times m_E$.
2. Array X mapped to SPM: In this case, the total-cost has two parts:

$$Total\_Cost = Transfer\_Cost + Access\_Cost$$

$Transfer\_Cost$ is the total cost of moving relevant parts of X from external-memory to SPM (or vice versa). Suppose that the transfers to SPM of the relevant portion of array X is done at loop-level k, i.e. inner to loop-nest $[i_1, i_2, \ldots, i_k]$ but outer to the loop-nest $[i_{k+1}, i_{k+2}, \ldots, i_n]$. Therefore, there will be in total $I_1 \times I_2 \times \cdots \times I_k$ number of transfers. We model each of those transfers as composed of N number of sub-transfers. For each sub-transfer, the DMA is programmed to move Q contiguous elements to the SPM from the external memory. A DMA-transfer incurs a fixed startup cost, C, and a cost that is

proportional to number of elements, ℓ, that are transfered [Kan04b], i.e. $DMA\_transfer\_cost = C + \ell t$. Therefore, each sub-transfer has a cost $C+Qt$. And so, $Transfer\_Cost = (C+Qt) \times N \times I_1 \times I_2 \times \cdots \times I_k$.

$Access\_Cost$ is the cost of all the readings/writings to the copy of X on SPM. Typically, only a small relevant part of X would be present on SPM at anytime. Let m_S be the cost per access for a word in the SPM. Therefore: $Access\_Cost = I_1 \times I_2 \times \cdots \times I_n \times m_S$.

7.6 SPM mapping algorithm

7.6.1 Illustrating example

We explain our algorithm, firstly, with an example. Figure 7.2 (left-column) shows a four-level nested-loop with $\mathbf{I} = [\ i_1 \quad i_2 \quad i_3 \quad i_4\ ]^T$. In this loop, array $A[100][200]$ is referenced as $A[\quad B[i_2] + i_3 \quad][\quad C[i_1] + i_4 \quad]$.

We can choose to do transfers of A, to the SPM, at five possible points. For instance, we could make a copy just before the start of loop i_4. That point is labeled as $migration\text{-}point\ m_4$ in the figure. At the point m_4, i_1, i_2 and i_3 each have a fixed value, temporary albeit. Starting at row $B[i_2] + i_3$ and column $C[i_1]+2$, the next four locations in that row can be copied to the SPM. These five locations, therefore, form a copy-candidate. The SPM-Mapping (third) column in Figure 7.2 shows how much of the SPM gets filled with that copy. The number of times this transfer would happen equals the number of times the program comes to $migration\text{-}point\ m_4$. That is, $50 \times 50 \times 80 = 200{,}000$ times. The $Total\_Cost$ at m_4 is:

$$Total\_Cost = \underbrace{(C + 5t) \times 200{,}000}_{Transfer\_Cost} + \underbrace{50 \times 50 \times 80 \times \times 5 \times m_S}_{Access\_Cost}$$

Rather than m_4, we could choose $migration\text{-}point\ m_3$. In that case we have to transfer 5×80 elements of A, starting at row: $B[i_2] + 1$ and column: $C[i_1] + 2$. But then, we have to do it only 50×50 times.

If we next try to do the copying at $migration\text{-}point\ m_2$, we must copy the entire 100 rows for the column $C[i_1] + 2$ to column $C[i_1] + 6$ (inclusive). The transfer will have to be done only 50 times. As pictorially illustrated in the Figure 7.2, with a slight increase in copy size we get an enormous reduction in the number of transfers (from 250, down to 50). That leaves us with two additional $migration\ points$: m_1, where we attempt to copy the entire array, but it is not possible because of limited SPM size; and m_5, where it is cheaper to access the external memory directly.

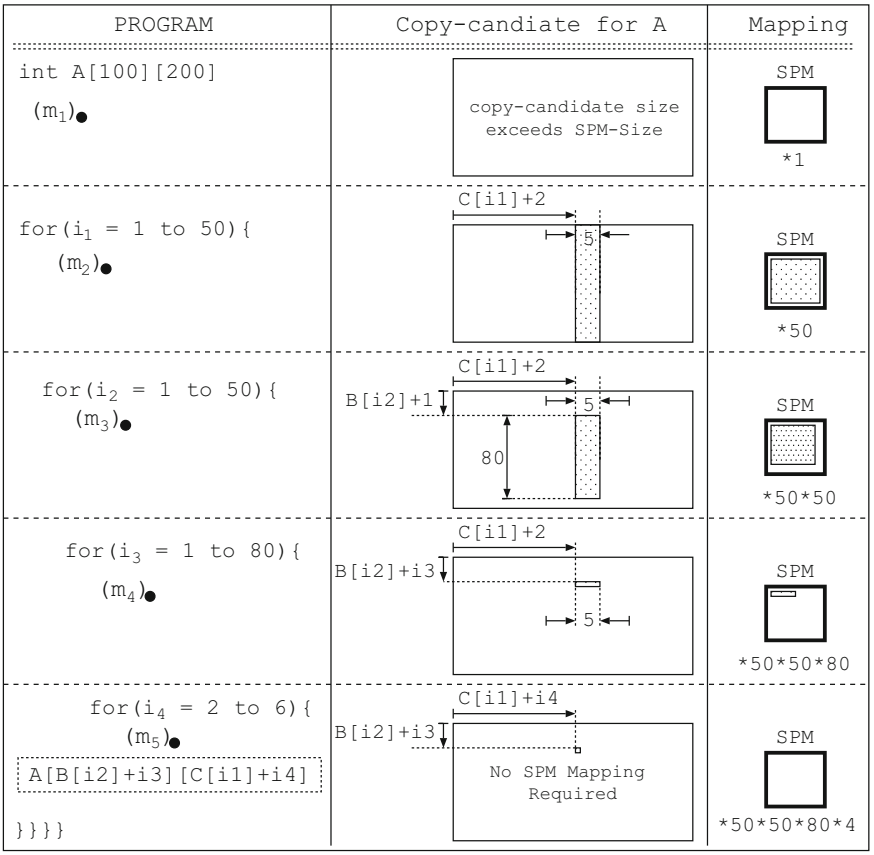


Figure 7.2: Trade-off between number of transfers and the size of each transfer

For each of the $migration\text{-}points$ above, we can compute the $Transfer\_Cost$. We chose the point that has the least cost. Note that $Access\_Cost$ is independent of the $migration\text{-}point$.

7.6.2 Search-space exploration algorithm

Consider a loop-nest with $\mathbf{I} = [\ i_1 \quad i_2 \quad \cdots \quad i_n\ ]'$. In our algorithm, we will need to describe subsets of the iterators. The set of all iterators is denoted as $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$. We define $\mathcal{I}_k$, a subset of $\mathcal{I}$, as $\{i_k, i_{k+1}, \ldots, i_n\}$; e.g. $\mathcal{I}_2 = \{i_2, i_3, \ldots, i_n\}$.

The m-dimensional, indirectly indexed array X is accessed inside the loop-nest $\mathbf{I}$ described above. The size of X is $D_1 \times D_2 \times \cdots \times D_m$. X is referenced

as $X[E_1][E_2]\cdots[E_m]$, where each E_p is a linear expression of iterators and indirectly indexed arrays ($U_{p,q}$). That is,

$$\Re_X(\mathbf{I}) = \left[\begin{array}{cc} \sum_q U_{1,q}\langle\Re_{U_{1,q}}(\mathbf{I})\rangle & \sum_q U_{2,q}\langle\Re_{U_{2,q}}(\mathbf{I})\rangle \quad \cdots \end{array} \right]^T + R_X\mathbf{I} + \mathbf{o}_X$$

We refer to $U_{p,q}$'s as the indirectly indexed arrays at the $first\text{-}level$. The iterators over which X is regular in the pth dimension is represented as α_p. Similarly, the iterators over which X is irregular in the pth dimension is represented as β_p.

First, we transform the loop-nest to improve locality, based on the computed reuse vector as described in [Abs06]. Next, starting from the innermost loop (i_n), we traverse to the outermost loop (i_1), computing the $Transfer\_Cost$ for each $migration\text{-}point$. The $migration\text{-}point$ m_k is located inner to the loop of iterator i_{k-1} but outer to loop of i_k (see Figure 7.2). Note that, during the execution, each time the program reaches m_k, the iterators in the set $\mathcal{I} - \mathcal{I}_k$ have a fixed valid value.

Our main data-structure for the algorithm is $config$. It has three components: (1) An array called $range$, (2) A number k which specifies the $migration\text{-}point$ m_k, and (3) The $Transfer\_Cost$ for the $migration\text{-}point$ m_k. The element $range[p]$ contains the lower and upper limits of the pth dimension that must be transferred. As seen in Algorithm 1, the search-space is traversed via two loops. The outer loop, $k = n, \ldots, 1$, is over the $migration\text{-}points$, starting from m_n (inner-most) and moving up to m_1. The second loop, $p = 1, \ldots, m$, is over the dimensions of the array X. For a given m_k, and a particular dimension p, our focus is to: find the range of the pth dimension of X that must be transferred to the SPM, assuming that the $migration\text{-}point$ has been fixed as m_k.

We, first of all, check if any of the iterators in $\mathcal{I}_k$ are present in the irregular set (β_p). Recall that iterators in β_p are those used to indirectly index X in the pth dimension. Also, iterators in the set $\mathcal{I}_k$ are those not having a definite fixed value at m_k. They cover a range. Therefore, if any iterator in $\mathcal{I}_k$ is also present in β_p, we cannot know the exact locations in pth dimension of X that would be accessed. Therefore, if $\mathcal{I}_k \cap \beta_p \neq \phi$ (ϕ is the null-set), we copy the entire range $(0, \ldots, D_p - 1)$ to the SPM (see Algorithm 1).

If $\mathcal{I}_k \cap \beta_p = \phi$, then we check the regular set, α_p, which represents iterators generating regular access in the index-expression E_p. Iterators in $\mathcal{I} - \mathcal{I}_k$ have a fixed value at m_k. Therefore, the free-iterators in E_p – those representing not one particular value at m_k, but a range – is the set $\alpha_p \cap \mathcal{I}_k$. These iterators in E_p are analyzed to determine the range of the pth dimension of X that should be transferred to the SPM. If $\mathcal{I}_k \cap \alpha_p = \phi$, then only a single point in this dimension is to be transferred.

Algorithm 1 An algorithm for mapping indirectly indexed arrays to scratch-pad.

Function: Find optimal $mapping$ to scratchpad for indirectly indexed array

for $k = n$ downto 1 **do**
  for $p = n$ downto 1 **do**
    if $I_k \cap \beta_p \neq \phi$ **then**
      $range[p] \leftarrow (0 \ldots D_p - 1)$
    else if $I_k \cap \alpha_p \neq \phi$ **then**
      $range[p] \leftarrow$ compute range based on variables $I_k \cap \alpha_p$ ...
      ... in E_p, treating $I - I_k$ and arrays as constants
    else
      $range[p] \leftarrow$ a single point
    end if
  end for
  $c \leftarrow transfer\_cost(range), config[k] \leftarrow \langle range, k, c \rangle$
end for
return $config$ with the minimum cost

For each m_k, a $config_k$ is generated. It contains the $range$ information and associated $Transfer\_Cost$. If the sizes in $range$ amount to more than the allocated SPM-size, the cost is set to infinity (not shown in Algorithm 1). We choose the optimal config as one with the minimum cost. This is compared with the $No\ SPM\text{-}Mapping$, i.e. direct access from external memory.

The data-layout of X should be changed so that the dimension that has the largest range is the $fastest\ changing\ dimension$ [Kan99]. This data-layout will result in the least number of DMA-transfers. In other words, based on the dimensions of the hyper-cuboid that must be transferred, the layout must be such that the longest edge is stored contiguous in memory.

Now we consider the case when the range of values assumed by the elements of (at least some) indirectly indexed arrays are provided to the algorithm. In real embedded application, this is usually possible because, even for data-dependent parameters, the designers typically apply some bounds. With that information, in Algorithm 1, for the case $\mathcal{I}_k \cap \beta_p \neq \phi$, we do not automatically extend the range to full-length $(0, \ldots, D_P - 1)$. Instead, arrays, such as $U_{p,1}, U_{p,2}, \ldots$, which indirectly index X at the $first\text{-}level$ in the pth dimension, are treated as variables that can assume any value in the specified range. Also, iterators in E_p belonging to the set $\mathcal{I}_k \cap \alpha_p$, i.e. iterators whose value is not fixed at m_k, are also treated as variables. Treating the rest of the iterators in E_p as constants, the total range is computed.

7.7 Experiments and results

The output program generated using our technique is standard C code, with API calls for DMA transfers. The optimized program is compiled and executed on the cycle-accurate ARM SystemC based simulator [Ben05b]. In the simulator, all components have incorporated timing and power models and hence, the simulator is able to provide accurate energy and performance numbers. We present results on key kernels from the following benchmarks:

- QSDPCM (Quad-Tree Structured Difference Pulse Code Modulation) which is an inter-frame compression technique for video images [Sto98].
- PEGWIT: A Public-Key Encryption algorithm from Media-Bench [Med97]. It has multiple levels of indirect indexing. Its kernel comprises of ordinary operations such as multiplication and addition, but because the numbers involved are elements in the Galois-Field, these operations take several cycles per operations.
- Cavity-Detector: A medical imaging algorithm.
- AC-3 Encoder: Audio compression algorithm [Tod94] (also known as Dolby-Digital). The bit-allocation part has several two level array-indexing.

For the experiments we used the following parameters for SPM mapping evaluation: C (constant factor in DMA-Transfer) $= 15$, t (cost per element, for the transfer) $= 1$, m_E (cost for data in external memory) $= 5$, and m_S (cost for data in SPM) $= 1$. These are typical values used by other researchers [Kan04b].

Figure 7.3 shows the energy consumption for each application. It includes the energy of the external memory, on-chip memory (scratchpad of cache) and the DMA (direct memory access device). For each application, Figure 7.3 has a group of three columns. The first column shows the energy value for the original program. No SPM mapping ($Non\text{-}Opt$) was done but the arrays were accessed through the cache. The second column shows the execution-time when all the arrays with only regular access (directly-indexed) were mapped to the SPM, in the best possible way ($Opt\text{-}Reg$). Arrays not mapped to the SPM were put in the cacheable-section of the external memory. The third column shows the execution time when both regularly and irregularly accessed arrays were allowed to be mapped to the SPM, using our technique ($Opt\text{-}Irreg$). Figure 7.4 shows the execution time for the three versions. Clearly, SPM mapping improves execution time and, even more, the energy consumption.

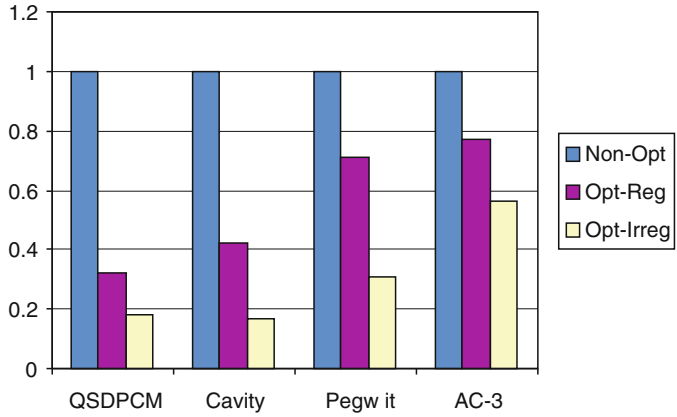


Figure 7.3: Total energy consumption of each application. Results are provided for three versions - 1. $Non\text{-}Opt$, 2. $Opt\text{-}Reg$, 3. $Opt - Irreg$

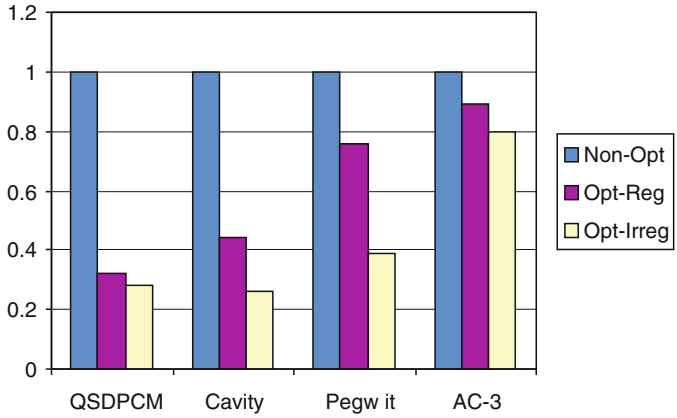


Figure 7.4: Total execution time for each application and each version

Next, we study the impact of different scratchpad sizes (see Figure 7.5). For this, we take the $Opt\text{-}Irreg$ in Figures 7.3 and 7.4 as the reference (i.e. as one) and show the improvements (or otherwise) in performance as we scale the scratchpad size. For QSDPCM, a bigger SPM means greater exploitation of reuse for the indirectly indexed arrays such as $image$. The motion vector value, though run-time dependent, has a limited range (search is limited to a certain range in the application). This information is communicated to the mapping algorithm via constraints. This information is subsequently used to increase SPM usage. For applications like cavity and pegwit, increasing the SPM size beyond a certain point has negative impact because larger SPM size implies more energy per access. If that loss is not amortized by higher reuse,

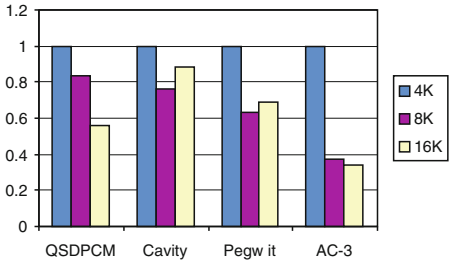


Figure 7.5: Energy performance for different scratchpad sizes

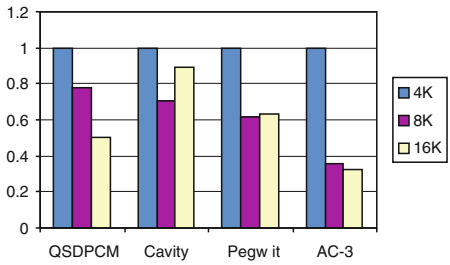


Figure 7.6: Execution time for different scratchpad sizes

then we may see a decrease in performance with increase in memory size. Figure 7.6, similarly, shows the execution time for different SPM sizes.

7.8 Handling dynamic data structures on scratchpad memory organisations

In the remainder of this chapter, we will study the problem of access and layout locality optimization, and mapping to scratchpad memory, of data that exhibit dynamism in their data organization or exhibit dynamism in their memory referencing.

Dynamism in data organization is exhibited when either the size of the data structure changes rapidly, through insertions and deletions, or the linkage and position between the data items evolve with time. This kind of behavior is typically seen in pointer based structures such as trees, heaps, graphs and linked lists, but is definitely not limited to them, as we will see in the subsequent sections. Also, it is not necessary that a pointer data structure will always exhibit dynamic data organization. Dynamism in memory referencing, on the other hand, is exhibited when the data access pattern seems irregular,

unpredictable and is not easily amenable to compile-time analysis. Examples of this are indirect array indexing (as discussed earlier in this chapter), hashing and dictionary search.

Optimizing locality of data structures, that exhibit dynamism in data organization or in referencing, can become extremely difficult. The reason is that dynamism hinders reuse analysis. When there is dynamism in the data organization, the reuse pattern of the data items being accessed change rapidly with time. For example, when an AVL-tree is rebalanced, nodes that are down below near the leaves may come up toward the root. The nodes near the root are reused or accessed more often. However, with the next rebalancing new nodes may emerge at the top throwing the old ones at the top to down below. Therefore, which nodes would be reused depends entirely on the data that triggers the reorganizations. When the referencing is dynamic, e.g. based on input data, it can be difficult to know beforehand the next data item that would be accessed. Therefore, locality optimization in such situations work better at the algorithmic level rather than at code transformation level.

Dynamism in data organization and referencing is not only a problem for locality optimization, but also for mapping to scratchpad memory (SPM). Using SPM makes sense only when we know which are the data items that would be accessed next. Then those data items can be moved to the SPM. With dynamic referencing, it is difficult to know what will be accessed next, and so it can be difficult to know what to move into SPM and what to move out. With respect to dynamic data organization, using a SPM can create problem because data items currently on SPM may lose priority and become less likely to be accessed in the future. Adopting to the new structure could then mean moving a lot of items out of SPM and bringing new data items into it. The high frequency of transfers necessary just to match with repeated changing of data organization can then undo the benefit of SPM over cache.

7.9 Related work on dynamic data structure access

Banakar et al. [Ban02] performed a detailed study of the energy and area advantage of the SPM over the cache. On a per access basis, they found that a 2 KB, 2-way cache consumed 4.57 nJ, while a 2 KB SPM consumed only 1.53 nJ. Initial work by Panda et al. [Pan97] on utilizing the SPM focused on scalars and highly used, small-sized arrays. Dominguez et al. [Domi05] explain a technique for mapping dynamic data structures, e.g. linked lists and trees, to SPM. The size allocated in the SPM to each set, e.g. nodes of a particular tree, is made proportional to the overall number of access to that set. They do not compare between SPM and cache but focus just on finding the best allocation on the SPM for the different dynamic objects.

For the cache, itself, there have been numerous studies to estimate its performance for regular and non-regular applications. Several studies have tried to quantify the cache performance by summarizing or analyzing actual memory access trace [Aga89, Sin92]. From analytical comparison perspective, however, trace analysis is not fruitful. On the other hand, the Independent Reference Model (IRM) of Rao [Rao78, Kin71] is more suited to our purpose of analyzing cache behavior for comparison with SPM. This model was recently extended [Lad99, Fix02] to algorithmic analysis. Rao's equations assume a fixed data-layout. However, results in this chapter allow conclusions to be drawn across all possible layouts.

7.10 Dynamic referencing: locality optimization

In this section, we will study locality optimization and mapping of data structures that have dynamism in access pattern, for example due to continuously changing input data. Let us start with a very interesting example. The function $search$ in the program section below, does a spell-check by performing a search of the given $word$, of n letters, in its dictionary. If the word is found in the dictionary, then the spelling is considered to be correct.

```
typedef struct node{ //trie data-structure
  struct node *next, *down; char letter;
}Node;

bool search(Node *nptr, char *word, int n){
  for( i = 0 ; i < n ; i++ ){
    while(nptr && (nptr->letter < word[i])){
       nptr = nptr->next;
    }
    if(nptr){
       if(nptr->letter == word[i]) //word found
         nptr = nptr->down;        //move to next word
       else  return false;         //word not found
    }else  return false;           //word not found
  }
  return true;                     //word found
}
```

The dictionary is implemented using a trie. A trie is an ordered tree data structure that is used to store associative arrays, where the keys are strings. Unlike a binary tree, in a trie the full key, which is a search-word in this case, is not stored in any node. It is the position of the node in the trie that denotes

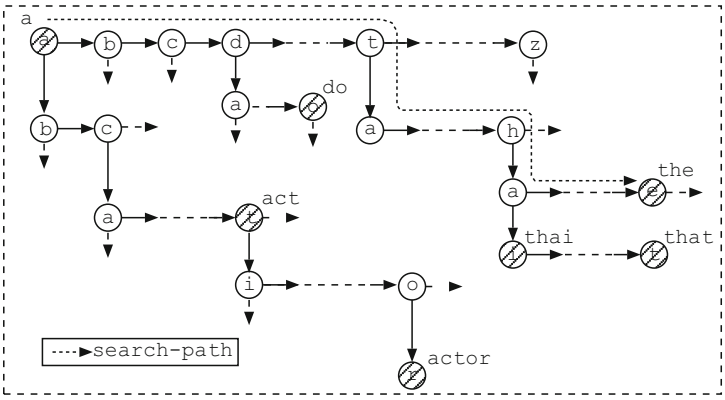


Figure 7.7: Spell-checker implemented with a trie data structure for fast reTRIEvals. Search-path to the word *the* is delineated with a dotted line

what key it represents. Also note that all the descendants of a node have a common prefix. The trie data structure is the most common approach to implementing dictionary [Kuk92]. The spell-checker is shown in Figure 7.7. The search-path to the word *the* is delineated with a dotted line in the figure.

Our goal is to optimize the locality of the memory accesses in the spell-checker. There is no obvious access reordering possibility in this application, so we can only optimize layout locality. With respect to layout locality, we have two choices for the L1-memory. We can map our spell-checker, under software control, on SPM. Or, we could simply use a data-cache. For the case of data-cache, since the tracking and migration of relevant data items to and from the cache is handled by hardware, no changes are required on the code. For the case of SPM, however, additional instructions are necessary to place the important nodes on SPM. The rest of the nodes remain in external memory from where they are directly accessed.

Access to the nodes mapped to SPM will be quick and energy efficient. If the nodes are mapped intelligently such that most of the search is to nodes on the SPM, then the overall performance and energy efficiency can be expected to be high. The question is: Can the SPM really outperform the cache in such cases of dynamic and data-dependent referencing? How do we decide what to map to SPM? Note that once the dictionary has been built, the data structure does not undergo changes anymore. Therefore, this is a case of dynamic referencing with static data organization. Now, we we could just try to find out empirically if spell-checker mapped to SPM outperforms the cache. However, that will not tell us much about how the next application with similar characteristics would behave. Therefore, let us take a more theoretical approach to converge on the right answer. Following that, we will naturally cross-check it with empirical results.

7.10.1 Independent reference model

Computations such as matrix multiplication and digital filtering generate *memory reference strings*, i.e. sequence of data memory addresses, that are regular and input-independent, and can be determined without even running the program. This makes mapping to SPM easy, and as such the SPM is able to outperform the cache for all these types of computations [Kan04b].

Indirectly indexed arrays posed a problem, for locality optimization and mapping to SPM, and was discussed in Chapters 4 and 5, respectively. Note that in the case of indirectly indexed arrays, we were able to avoid part of the uncertainty or irregularity by tackling only those indirectly indexed arrays and only those indexing arrays that were not written to in the same loop-nest in which they were read. Efficient mapping to SPM was then achieved by dividing the problem into separate decisions for each dimension of the indirectly indexed array. The techniques of indirectly indexed arrays cannot be applied in the same way to applications like the spell-checker. Moreover, instead of just proposing some technique for SPM mapping, firstly we want to determine theoretically, if SPM is indeed a good solution for these kind of applications, or is the cache better.

Dynamic applications, with dynamic referencing, generate reference strings that are irregular and input-dependent. Therefore, predicting SPM and cache performance, i.e. hit-rate, with the techniques used for the regular case turns out to be unwieldy and complicated. Therefore let us use statistical methods [Tri02] to model the SPM and the cache behavior. This will lead us to determine when a good mapping to SPM is possible and when it is not. We start by characterizing the reference string using the Independent Reference Model (IRM) [Lad99][Rao78]. The basis of this model is that we cannot know with certainty the next data that will be accessed, since it is input dependent, but we can presumably describe it with a certain probability function.

Consider a set of objects, or data items, $O_1, O_2, \ldots, O_n$ and a set of corresponding *access probabilities* $p_1, p_2, \ldots, p_n$. The reference string, i.e. the string containing the list of objects that are referenced at various times, can be denoted as $r_1, r_2, \ldots, r_t, \ldots, r_N$. Here r_t is the object referenced at time t. Under IRM:

$$Pr[r_t = O_i] = p_i\,,\ 1 \leq i \leq n\,,\ t > 0$$

That is, the probability that O_i is accessed at time t is p_i. Though this model does not take into consideration the correlation between the accesses, studies [Jel04] show that the behavior modeled assuming $independent\ reference$ gives results that are very close to those obtained using models that do indeed incorporate such correlation. We will also show this with empirical results

which reconfirm that IRM is indeed able to accurately predict SPM and cache performance, for data-structures such as trees where the access-patterns are clearly data-dependent and correlated.

Now, let us use this model to estimate the cache hit-rate for a given set of access probabilities. Suppose we have a cache with just one block and so it contains only the last object accessed. It can been shown [Kin71] that the states of this cache forms a homogeneous Markov chain, where each state S_i is defined as having the object O_i inside the cache block. The equilibrium probability of state S_i equals p_i [Rao78] and hence, the probability that object O_i is inside the cache, at anytime, equals p_i. Therefore, if object O_i is accessed, then the probability that the access results in a hit (object found in cache) equals p_i. Averaging the hit-rate across all possible accesses, we get the expected hit-rate η for a cache of size one as:

$$\eta = \sum_{i=1}^{n} p_i^2 \tag{7.2}$$

Embedded systems usually contain caches with low associativity. Therefore, results that we derive are in the context of the direct-mapped cache (DM-Cache). However, caches with low associativity perform similar to the DM-Cache [Rao78]. We will later present empirical results that confirm this.

The expected hit-rate for a DM-Cache of more than one block can be determined by placing the objects into disjoint groups. Assume that the DM-Cache contains m cache blocks, and each block holds only one object. Let G_i denote the set of objects which map to cache block i. Out of the n objects $O_1, O_2, \ldots, O_n$, assume, for simplicity, that $k = n/m$ objects map to each cache block. Objects in G_i are denoted as $O_1(i), O_2(i), \ldots, O_k(i)$, and the corresponding probabilities as $p_1(i), p_2(i), \ldots, p_k(i)$, respectively. Let $D_i = p_1(i) + p_2(i) + \cdots + p_k(i)$. The following result, from Rao [Rao78], gives the expected hit-rate of a DM-Cache:

$$\eta_{\text{DM}} = \sum_{i=1}^{m} \frac{1}{D_i} \sum_{j=1}^{k} p_j^2(i) \tag{7.3}$$

Essentially, a DM-Cache behaves as m disjoint, fully-associative caches, each of size one. So, Eq. 7.2 can be applied to each cache block, but with conditional probabilities $p_j(i)/D_i$. The overall hit-rate equals the weighted sum of hit-rates of each individual block. The weight for a block equals the probability that the next access would be to that block. For block i, this equals D_i.

Let us now illustrate how access probabilities can be determined for different data structures. A linked-list is shown in Figure 7.8. Each node contains a key and some data. The search for a node, with a certain key, starts at the head and continues till that node is found. Each key is equally likely to be

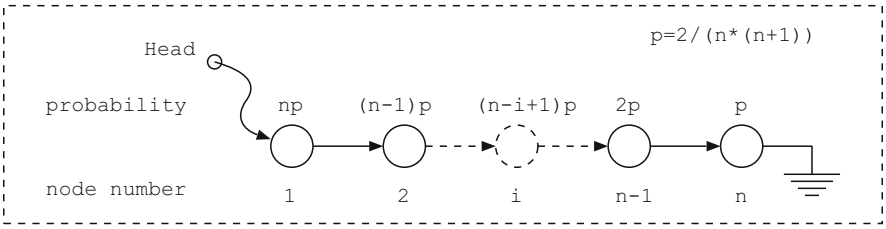


Figure 7.8: Access-probabilities in a linked list, when each node is equally likely to be the target of the next search. A search stops when target is found

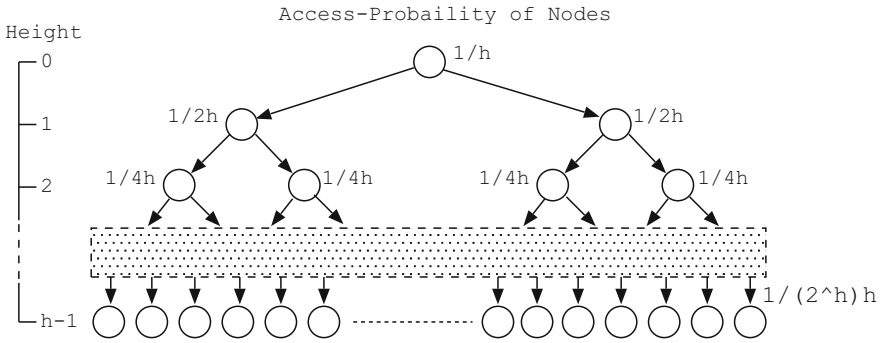


Figure 7.9: Access-probabilities of nodes in a binary tree

the target of the next search. Now, if the nth node has access probability p, then the $(n-1)$th node has access probability $2p$. Reason: the $(n-1)$th node is referenced when the nth node is the target of a search, and it is also referenced when it is, itself, the target of a search. Probabilities for all the nodes is shown in Figure 7.8. It is also possible to assign probabilities in an application-specific way or based on profiling.

Next, consider a binary search-tree. A search starts at the root and proceeds downward to the leaf-nodes. From any parent, the search-path has an equal chance of moving to the left child-node or to the right child-node. Therefore, if the access probability of a parent is p, each child-node has access probability $p/2$. Figure 7.9 shows the access probabilities of the nodes in a binary tree. Note the exponential decline in access probabilities.

7.10.2 Comparison of DM-cache with SPM

In this section, we will make a theoretical comparison of SPM and DM-Cache performance using the Independent Reference Model. The purpose

is to determine if the hit-rate of DM-Cache exceeds the SPM. That is, for objects referenced dynamically, in a data-dependent way, which could be approximated by a probability distribution function.

To recapitulate, we have n objects $O_1, O_2, \ldots, O_n$ with access probabilities $p_1, p_2, \ldots, p_n$, respectively. Without loss of generality, let $p_1 \geq p_2 \geq \cdots \geq p_n$. To maximize the SPM hit-rate, objects $O_1, O_2, \ldots, O_m$, where m is SPM size, must be placed on SPM. The rest of the objects remain in the memory, from where they are accessed directly. Hence, each access to O_i, where $i > m$, constitutes a miss. The expected hit-rate of this optimal SPM mapping is:

$$\eta_{\text{SPM}} = \sum_{i=1}^{m} p_i \tag{7.4}$$

Let us now compare this with the hit-rate of a DM-Cache, also of size m, using Eq. 7.3. 4As before, objects $O_1(i), O_2(i), \ldots, O_k(i)$, with access probabilities $p_1(i), p_2(i), \ldots, p_k(i)$, respectively, map to cache block i. Again, without loss of generality, let $p_1(i) \geq p_2(i) \geq \cdots \geq p_k(i)$. Although we assume that exactly $k = n/m$ objects map to each block, it is not a limitation of the proof, but is done so as to simplify the notation.

Since $D_i = p_1(i) + p_2(i) + \cdots + p_k(i)$ in Eq. 7.3, we have $p_1(i) = D_i - \sum_{j=2}^{k} p_j(i)$ Now, Eq. 7.3 can be rewritten as:

$$\begin{aligned}
\eta_{\text{DM}} &= \sum_{i=1}^{m} \frac{1}{D_i} \sum_{j=1}^{k} p_j^2(i) \\
&= \sum_{i=1}^{m} \frac{1}{D_i} \left[p_1^2(i) + \sum_{j=2}^{k} p_j^2(i) \right] \\
&= \sum_{i=1}^{m} \frac{1}{D_i} \left[p_1(i) \left(D_i - \sum_{j=2}^{k} p_j(i) \right) + \sum_{j=2}^{k} p_j^2(i) \right] \\
&= \sum_{i=1}^{m} p_1(i) - \sum_{i=1}^{m} \frac{p_1(i)}{D_i} \sum_{j=2}^{k} p_j(i) + \sum_{i=1}^{m} \frac{1}{D_i} \sum_{j=2}^{k} p_j^2(i) \\
&= \sum_{i=1}^{m} p_1(i) - \sum_{i=1}^{m} \frac{1}{D_i} \sum_{j=2}^{k} \left(p_1(i) p_j(i) - p_j^2(i) \right) \\
&= \sum_{i=1}^{m} p_1(i) - \sum_{i=1}^{m} \frac{1}{D_i} \sum_{j=2}^{k} p_j(i) \left(p_1(i) - p_j(i) \right)
\end{aligned} \tag{7.5}$$

Now, since $p_1(i) \geq p_j(i)$ for all $1 < j \leq k$, in Eq. 7.5 each expression $p_j(i)\big(p_1(i) - p_j(i)\big)$ is always a positive number. Therefore, the second term in in Eq. 7.5 is a positive number and so:

$$\eta_{\text{DM}} \leq \sum_{i=1}^{m} p_1(i) \tag{7.6}$$

The expression $\sum_{i=1}^{m} p_1(i)$ in Eq. 7.6 attains its maximum value when the objects $O_1(1), O_1(2), \ldots, O_1(m)$, with probabilities $p_1(1), p_1(2), \ldots, p_1(m)$, respectively, are any permutation of the objects in the set $\{O_1, O_2, \ldots, O_m\}$. Note that $O_1, O_2, \ldots, O_m$ are the objects with the highest access probabilities among all the n objects. Therefore, the highest value attained by η_{DM} is $\sum_{i=1}^{m} p_i$. Comparing this with Eq. 7.4, we conclude that $\eta_{\text{DM}} \leq \eta_{\text{SPM}}$.

Therefore, the result of IRM analysis is that the SPM with an optimal mapping can always outperform the DM-Cache. We must again emphasize that this result holds, firstly, if the access pattern can be well approximated using access probability. Secondly, the subtle assumption has been that the access probabilities do not change over time. This assumption is almost true if the data organization does not change. We will elaborate this further under dynamic data organization discussions.

7.10.3 Optimal mapping on SPM–results

Let us now verify the theoretical conclusion with experimental results. Let us start with the simple example of key-search on a linked-list. Search begins at the head and stops when the target is found. Each node is equally likely to be the target of the search. We computed the access probabilities of the nodes earlier, and had come to the, sort of obvious, conclusion that the nodes near the head of the list have the higher access probabilities. Therefore, the first m nodes are mapped to SPM.

Figure 7.10 plots the measured hit-rates for searches conducted on the linked list. In the experiment, the L1-memory size is 4KB, with cache block-size 16B. Each node is 16B. In the case of SPM, the first m (256) nodes from the head of the list were placed on SPM. For the DM-Cache case, the first m nodes were placed in different cache blocks. From Figure 7.10 we see that SPM indeed outperforms the direct-mapped cache. The cache performance, with respect to SPM, worsens for increasing problem size due to increasing conflicts.

The *spell-checker* [Kuk92] was described in detail earlier in this chapter. At first, one might be tempted to map the spell-checker onto the cache because it involves data-dependent traversal of a pointer-based dynamic data structure. However, we will see that with a smart mapping the spell-checker actually

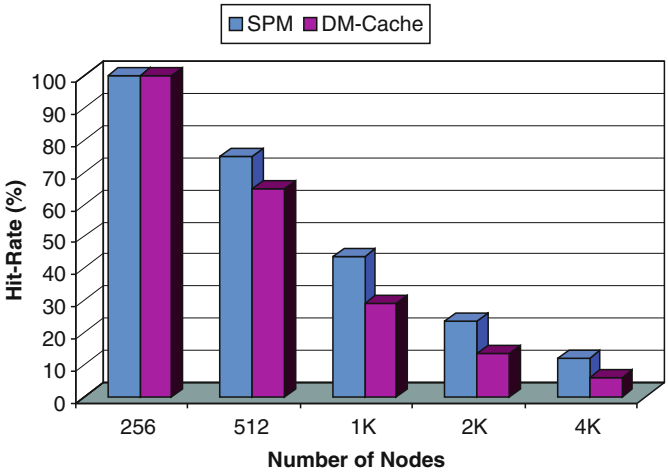


Figure 7.10: Hit-rates for key search on the linked list. Cache performance worsens, compared to SPM, with increasing problem size (number of nodes)

does better on SPM. The challenge is to determine which nodes have the highest access probability and then map them to SPM.

In our experiment, the trie contains over five thousand commonly used words. In addition to using a dictionary (Cambridge) to fill the words in the trie, we also added popular proper nouns. We then performed a *mock* spell-check with a large *training-essay* of several thousand words. The frequency of access to each node was then used to estimate the access probability. The estimated access probabilities were then used to map the most heavily accessed nodes to SPM.

The access probabilities were also then used to predict the hit-rates using Eqs. 7.3 and 7.4. For SPM, it assumes that the nodes with highest access probabilities are mapped to SPM, while for the DM-Cache case it assumes that they map to different cache-blocks. The actual hit-rates are measured by running the spell-checker over online newspaper articles (NYTImes and Washington Post) of more than hundred thousand words.

Figure 7.11 shows the predicted and measured hit-rates. The first observation is that the predictions, both for SPM and DM-Cache, is close to the measured values. Therefore, IRM using access probabilities is indeed able to model SPM and cache behavior for data-dependent traversals to a high degree of accuracy. The second observation is that, as proved in previous section, the SPM with optimal mapping does indeed outperform the DM-Cache – albeit by a small margin. However, since SPM is considerably more energy efficient than cache, the absolute gain in energy numbers is significantly higher.

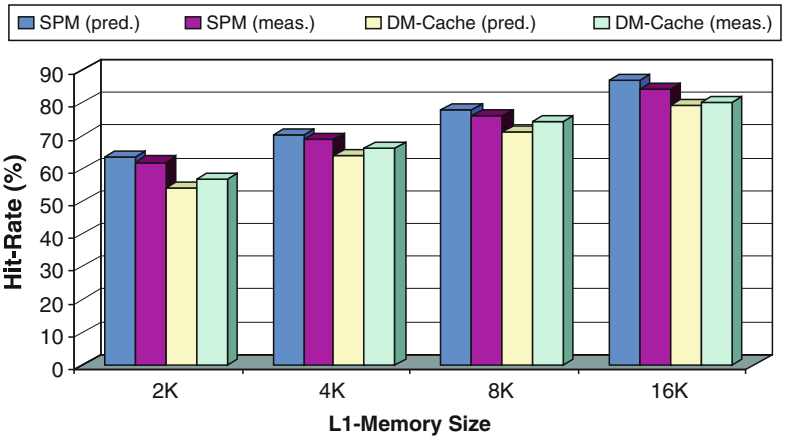


Figure 7.11: Spell-checker: Predicted (pred.) and experimentally measured (meas.) hit-rates of SPM and DM-Cache. Predictions are 2.6% accurate

Let us now compare the previous SPM and DM-Cache mappings with other mapping techniques. Typically, the trie is *grown* by inserting new words into it. In the first version SPM-1, the words are inserted into the trie in lexicographical order and the nodes are allocated space in the SPM on a first come first serve basis, till the SPM gets full. In the SPM-2 version, the trie is built by inserting the most commonly used words (e.g. *the*, *as*, *and*) first, and then inserting the remaining words. Therefore, words which are searched most often have their paths *almost* entirely on the SPM. We say *almost* because words inserted later-on could potentially add new nodes in the paths to the commonly used words.

Figure 7.12 shows the hit-rates for SPM-1 and SPM-2, and compares them with SPM-3 and DM-Cache. The SPM-3 version puts the nodes with highest access probabilities on SPM. Therefore, SPM-3 is identical to SPM (meas.) in Figure 7.11 but is shown again for comparison. DM-Cache in Figure 7.12 shows the hit-rate when no customized mapping of nodes is done. This is different from the experiment conducted for DM-Cache (meas.) in Figure 7.11 where nodes with highest access probabilities were mapped to different cache-blocks. As expected, the hit-rate values for DM-Cache in Figure 7.12 are less than that in Figure 7.11. From Figure 7.12, we therefore conclude that mapping using access probabilities can be superior compared to conventional mapping techniques.

Next, we study the impact of increasing associativity on the performance of the cache. In particular, we would like to see if set-associative caches can outperform the SPM. Figure 7.13 compares direct mapped (1-way) with 2-, 4- and 8-way set-associative cache. The SPM hit-rate (columns for SPM-3)

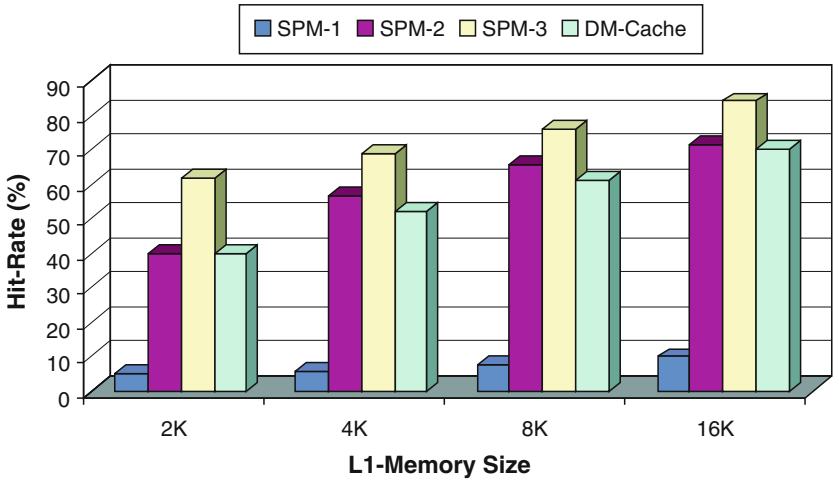


Figure 7.12: Spell-checker: Comparison of SPM with DM-Cache. Using access-probabilities (SPM-3) gives better results compared to conventional mapping schemes (SPM-1: lexicographic insertion, SPM-2: common words first).

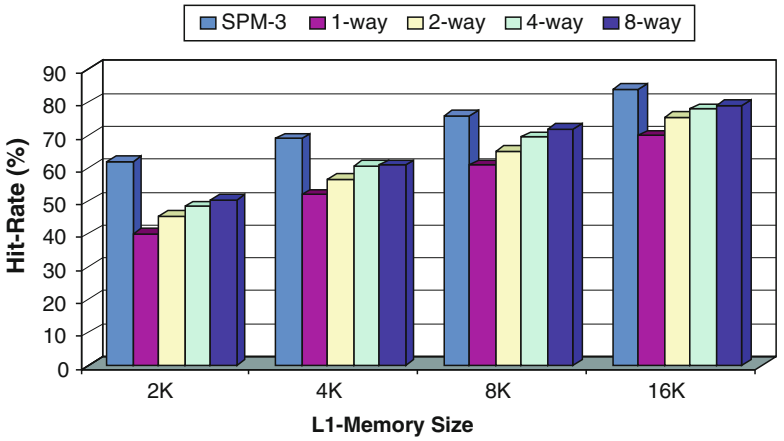


Figure 7.13: Spell-checker: SPM performance compared to 1-, 2-, 4- and 8-way set-associative caches. Increasing associativity does not improve performance.

are also shown for comparison. We see that increasing the associativity does not tilt the performance toward cache. Moreover, set-associative caches are power hungry and hence any performance gains comes at high energy penalty.

Let us now summarize our discussion on locality optimization for data structures referenced dynamically, primarily due to data-dependent access. We have seen, based on experimental results, that IRM can be used to model such referencing quite accurately. The model is a good approximation even for correlated accesses and associative caches. Using the model, we derived the result that the optimally mapped SPM, i.e. when the nodes with the highest access probabilities are mapped to SPM, always outperforms the DM-Cache. This can be a surprise to many since it is conventional to think that the cache is always good for dynamic situations.

The critical hurdle in attaining the optimal SPM mapping is that the high access probability nodes may not be easily identifiable. Recall that for the spell-checker it was not so straightforward to reach the optimal mapping. In the case of the tree we at least know the high access probability nodes. However, flat data structure such as graphs often pose difficulty with respect to acquiring knowledge about which nodes have high reuse and which nodes would be accessed together. For instance, applications such as Voronoi and Perimeter, in the Olden [Car96] benchmark, are almost impossible to map effectively on the SPM. Which nodes would be accessed heavily next is not known in advance. The cache performs much better with those applications. Similarly, the A*star algorithm, often used to find the shortest path between two points in the terrain, such as in the popular game Microsoft's Age of Empire, references the array containing information about the obstacles in a seemingly almost random manner. Post-analysis of the accesses reveals significant reuse of data, over short periods, but this information cannot be gleaned much in advance to actually exploit it.

7.11 Dynamic organization: locality optimization

In this section, we will study locality optimization and mapping to SPM, of data structures that exhibit frequent data organization during execution of the application. We will start with a very interesting example, Prim's algorithm for minimum spanning tree. In this case, even though the basic data structure is just an array, it actually embeds a complete binary tree. Every time the tree is restructured to enforce the heap property, the position of several data items in the array are reorganized. The number of reuse, for any data item, depends on its current location but the position keeps changing. However, as the heap is embedded in an array, this still offers some advantage in enabling a good locality solution. After MST, we will turn to even more dynamic cases where good locality solutions seems impossible to find.

7.11.1 MST using binary heap

Minimum spanning tree is useful in many applications including multimedia gaming. Prim's algorithm finds a minimum spanning tree (MST) for a given connected graph [Cor98]. The algorithm starts with a single node and adds one edge at a time, till all the nodes have been connected. At any time, during the construction of the tree, there is a set of nodes T already in the tree, and another set of nodes T' that are currently not in the tree. Each node in T' is assigned a weight that is equal to the cheapest edge connecting it to some node in T. Organization of the set of weights of the nodes in T', in such a way that the cheapest edge can be found quickly, is done using a binary heap.

A binary heap, such as in Figure 7.14, is a complete binary tree embedded into an array. Each node in the heap has a weight which is less than or equal to that of its two children. This implies that the node h with the minimum weight is always at the top (root). When h is removed, the last heap-node is moved to the top and then it *percolates* down to its new appropriate place. This structure can also be viewed as a priority queue, where the least weight has the highest priority and so is located at head of queue.

Based on our earlier analysis of the binary tree, we know that the nodes at the top of the tree have the highest access probabilities. Therefore, a good mapping to SPM constitutes putting the top part of the tree, as much as possible, on the SPM. In this case of binary heap embedded in an array, it literally means mapping the first m elements of the array on SPM. For the cache, the default mapping is already the best one.

Figure 7.15 shows the hit-rate of the MST algorithm on the SPM and DM-Cache, for a graph of over thousand nodes. Note that the SPM gives slightly

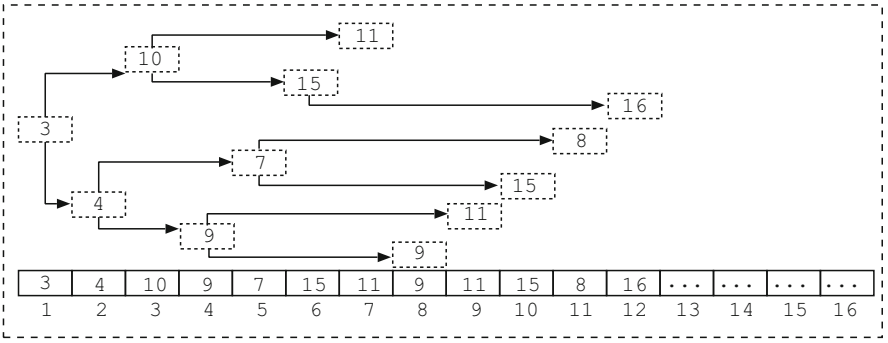


Figure 7.14: A binary heap implementation of the priority queue. It is used to construct the minimum spanning tree of a graph using Prim's algorithm

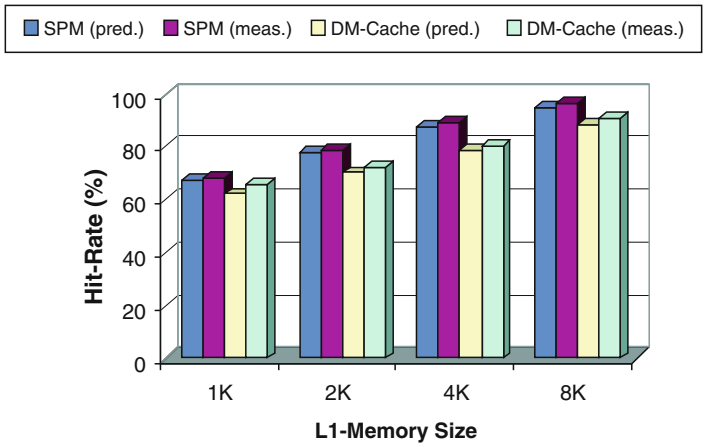


Figure 7.15: Minimum spanning tree: Comparison of hit-rates of SPM and DM-Cache. Predicted (pred.) values are from Independent Reference Model

better performance than the DM-Cache. Given that SPM consumes less energy per cycle, it is therefore clear that SPM should be the preferred choice for L1-memory.

Figure 7.15 shows both the predicted (pred.) and measured (meas.) hit-rates. In estimating the hit-rates (using Eqs. 7.3 and 7.4), we assume that a node percolating downward, ultimately reaches the bottom. This simplification has a certain degree of associated error. Secondly, note that the tree shrinks in size as nodes are removed from it. Therefore, the estimated hit-rate is the mean for the entire range. Observe that the predicted values are slightly below the measured values. The reason is that since some nodes do not percolate all the way down, the access probabilities are actually higher for nodes near the top of the heap as compared to what was assumed in our calculations. Figure 7.16 gives the hit-rates for different set associative caches. We see that increasing the associative does not have a big impact on the cache performance and the SPM still outperforms the cache.

7.11.2 Ultra dynamic data organization

The reason we were able to return high performance for the SPM, in the case of the minimum spanning tree algorithm, is because the tracking of the high access probability nodes was easy due to the typical array structure. The data items with the highest probabilities were always moved to the top of the array. Now, imagine if that was not the case and the high access probability nodes were dispersed all over the array. Then mapping becomes difficult since, for tracking what is on the SPM and what is not, it is always preferred to move a contiguous segment of the array to SPM.

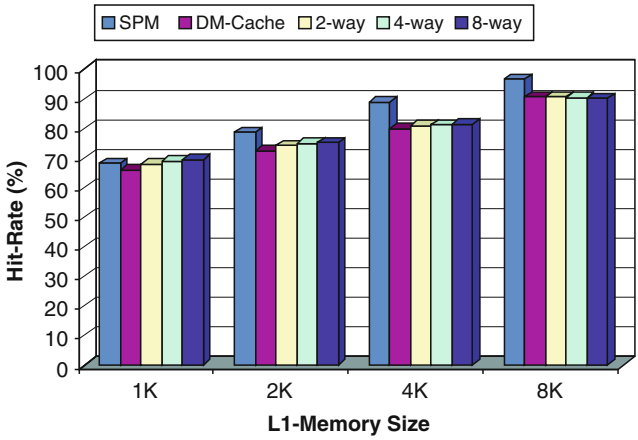


Figure 7.16: Minimum spanning tree: SPM performance compared with set-associative caches. Increasing associativity does not improve performance

Tree implemented using pointers pose a similar problem with respect to locality optimization. When a tree is rebalanced, nodes that were up near the top can move down or even be deleted. Nodes down below may be brought up with the next rebalancing. Assuming, quite reasonably, that the nodes near the top have high access probability, then to achieve good locality, nodes near the top should mostly be on the SPM. Firstly, they are accessed more frequently and, secondly, they are often accessed together (at least parent and immediate child), i.e. good for layout locality. However, to retain nodes at the top of the tree in the SPM, nodes continuously have to be moved in and out of the SPM. In the end, the extra cycles spent in achieving locality undo any gains resulting from the good locality.

We can explain the increased complexity of locality optimization, due to rapid data reorganization, from the perspective of the Independent Reference Model. When the access probabilities are static, it is often possible to identify the high access probability nodes and optimize their layout locality. As we have seen in the examples before, the access pattern is usually completely rigid and defined by the input data. Therefore, access locality improvement freedom is usually limited.

Now, when the data structure is continuously reorganized, the access probability of the data items change rapidly. Then, it is either too difficult to identify the top probability nodes, or continuous migration of data around to maintain top nodes on SPM becomes far too costly. The cache, on the other hand, due to the nature of the least recently used algorithm, can perform rather well in such a scenario of varying access probabilities. Let us

elaborate this further, because herein lies the answer to the question: In the case of dynamic data organization, is the cache performance really better than SPM?

The answer depends on a number of factors such as the application characteristics and the size of the working set. A major factor, which we discovered in our research, that influences the difference between cache and SPM performance, is the cumulative distribution function of the data-structure. Let us explain this. Without loss of generality, we can assume that the access probability of the data items are in the form $p_1 \geq p_2 \geq \cdots \geq p_n$. Let X be a random variable that takes on values from the set $\{1, 2, \ldots, n\}$. The cumulative distribution function (CDF), $F[x]$, is defined as the total probability that any object between O_1 and O_x , inclusive, will be accessed next. That is:

$$F[x] = Pr[X \leq x] = \sum_{i=1}^{x} p_i$$

Therefore, $F[1] = p_1$ and $F[n] = 1$. The CDF captures how the access probabilities are distributed over the entire n objects. The cumulative probability distribution function when each object has the same access probability is a linearly increasing function. On the other hand, for a data structure such as a binary tree, where the nodes at the top have high access probability while the nodes below have exponentially reduced access probability, the CDF is a logarithmic function (rising sharply and then increasing very slowly).

Having introduced the notion of CDF of access probabilities, we next need to define the concept of skewness in the CDF, in order to explain our major finding. We define a linearly increasing CDF as having zero skewness. Zero skewness is the result of all data items have the same access probability. When a few data items have a much higher access probability, compared to the remaining data items, the CDF becomes skewed. Therefore, CDF of a binary tree is more skewed than the CDF of a linked list. CDF of a linked list, on the other hand, is more skewed than the CDF of an array that is uniformly accessed. Note that $F[n] = 1$, and so if the access probability increases for a data item, it must reduce for some other item. Now, we give a result which connects the CDF to the expected hit-rate on a DM-Cache.

Theorem 1 *Hit-Rate of a Direct-Mapped Cache increases with increased skewness of the cumulative distribution function of access probabilities.*

We provide an intuitive proof for this. With a sufficiently large main-memory, each object X_i has an equal chance of being mapped to any of the m cache blocks. In total, there are m^n possible ways that n objects can map to m cache blocks. A very bad layout is when all the n objects map to the same block.

The probability of that happening is, however, only $m(1/m)^n$. This is rather small. With a highly skewed CDF, a few objects have the lion's share of the access-probability. The probability that these few objects would conflict with each other in the same cache block is low. Therefore, in the case of highly skewed CDF, the cache would return high hit-rate.

Now, consider the situation of the SPM. In a dynamic situation, we may not be able to identify the high probability nodes to move them to SPM. Therefore, in a dynamic situation of varying access probabilities, the SPM typically holds onto just an arbitrary subset of the total data structure. The probability that the important, high access probability, nodes reside on SPM, in fact, reduces with increasing skewness of the CDF. Therefore, when the access probabilities are unevenly spread across the objects, i.e. the CDF of the data structure is highly skewed, then the difference between SPM and cache performance, based on the above theorem becomes significantly different. Let us now check experimentally if this is indeed true.

Figure 7.17 compares the hit-rates of the SPM and DM-Cache for the application key search in the binary tree. The hit-rate of cache depends on the placement of the data in memory. When the placement is such that the nodes, specially high access frequency ones, conflict with each other then the hit-rate would be low. In a dynamic situation, where the binary tree is continuously reorganized, the application will have little control over the locations of the high access frequency nodes in the memory. Therefore, instead of showing the hit-rate for one particular placement, we simulated it over thousands of randomly selected placements. This will give us a good estimate of what to expect as the average behavior in the normal run of the program. For the SPM, we also randomly selected nodes to be placed on the SPM. Therefore, Figure 7.17 shows the frequency, with a probability density function (PDF), of the different hit-rates resulting from different placements. We see from Figure 7.17 that SPM gives hit-rate in the range 6–30% with the average around 13%. The DM-Cache, on the other hand, gives hit-rate in the

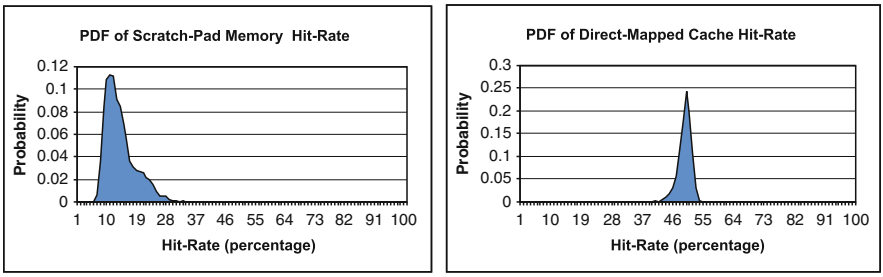


Figure 7.17: Hit-Rates of SPM and DM-Cache for key search on a binary tree. The PDF shows the results across different layouts of the data in memory

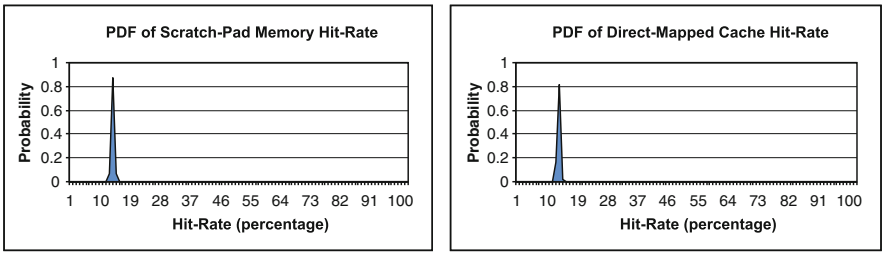


Figure 7.18: Hit-rates of SPM and DM-Cache for a uniform-random access of data. The variations in the hit-rate for different layouts is marginal

range 44–55% with the average around 51%. Therefore, the DM-Cache performance is far better than that of SPM. This is exactly what was predicted by the theorem above.

Figure 7.18 compares the hit-rate of the SPM and DM-Cache for a table that is accessed randomly. Both SPM and SM-Cache show similar hit-rates (around 15%) as predicted from the probability models. Moreover, note that the variation in hit-rate, for different data-layouts, is small. The reason is: since all objects have equal access probability, the sensitivity regarding which object conflicts with which other is small.

To conclude, the cumulative distribution function gives an estimate of the expected performance on the DM-Cache for dynamic data organizations leading to varying access probabilities. This expected hit-rate of DM-Cache can now be compared to the expected hit-rate if the data structure were to be mapped to the SPM. In a dynamic situation, where retaining nodes with high probability of access on the SPM requires additional overhead, we can use the CDF to predict if the DM-Cache would be able to do a better job.

7.12 Conclusion and key messages of this chapter

In this chapter, we have developed an algorithm for mapping indirectly indexed arrays efficiently to scratchpad memory. This algorithm tries to find out the appropriate portions of the array to be brought into the scratchpad at different time instants such that the total energy consumption is minimized. We have also demonstrated that our technique works well on real-life applications.

In addition, dynamic, or data-dependent, referencing to data structures such as tries, trees, heaps and linked lists can be modeled to a reasonable level

of accuracy using Independent Reference Model (IRM). We propose to use IRM to prove that scratchpad memories, with an optimal mapping based on access probabilities, can outperform the direct-mapped cache, irrespective of the layout influencing the cache behavior. This analytical result has then been verified with experiments. Increasing the associativity in the cache is shown not to improve the cache performance in any significant way. We also demonstrated that for dynamic data organization situations, where retaining nodes with high probability of access on the SPM requires additional overhead, the CDF can be used to predict if the DM-Cache would be able to do a better job.

CHAPTER 8

An Asymmetrical Register File: The VWR

Abstract

This chapter introduces one of the core contributions of the book which helps improve the energy efficiency of the register file. It presents a novel register file/foreground memory organization which is motivated across application, architecture and physical design abstraction layers. It is fully compatible with the energy-efficient scratchpad memory organisation that is proposed to be used for the large data storage in the previous chapter. It motivates across these different abstraction layers why the proposed register file is more efficient. It also shows that the proposed architecture is energy efficient over different DSP benchmarks.

8.1 Introduction

Register files have been known to be a notorious power consuming part of a processor architecture. It was already shown in Chapter 3 that there is a need for a comprehensive treatment of register files such that their power

F. Catthoor et al., *Ultra-Low Energy Domain-Specific Instruction-Set Processors*,
DOI 10.1007/978-90-481-9528-2_8,

consumption is reduced while still meeting all the realtime requirements of an application. In the previous chapter is has been shown that the energy-efficient storage of large data requires the maximal usage of scratchpad memory organisations instead of more conventional cache hierarchies. That is also true for non-regular and not fully static data structure access. In this way, the energy bottleneck moves even more to the foreground memory in the traditional multi-port register-file concept. Multi-ported data Register Files (RF) are one of the most power hungry parts of any processor, especially Very Long Instruction Word Processors (VLIWs) as shown in Chapter 3 and [Lam05]. On average every operation requires three accesses (two reads and one write) to the RF, which make them a very active part of the processor. Current architectures try to achieve a high performance by exploiting parallelism, and therefore perform multiple operations per cycle (e.g. Instruction Level Parallelism or ILP, as used in VLIW processors). This quickly results in a large port requirement for the register file, that is mostly implemented as a single/ centralized or distributed large multi-ported register file. A high number of ports has a strong negative impact on the energy efficiency of register files as well faces strong performance constraints for design. Traditionally, this problem is addressed through various clustering techniques [Rix00a] that partition (or bank) the RF. Data can then only be passed from one partition to another through inter-cluster communication [San01, Lap02]. However, as partitions get smaller the cost of inter-cluster copies quickly grows. In addition, the resulting register files are still multi-ported. For high energy efficiency, it is clearly preferable that the registers cells be single ported [Jan95].

Broadly speaking, from the application perspective, variables in an application can be of different types:

- Dynamic data types
- Input/Output arrays *with spatial* locality in access
- Input/Output arrays *without spatial* locality
- Intermediate arrays *with spatial locality* (Short[1] and Long lifetime)
- Intermediate arrays *without spatial locality* (Short and Long lifetime)
- Low life time intermediate scalar variables
- High life time intermediate scalar variables

These different variables are also shown in Figure 8.1. Dynamic data types are not in the focus of this book and it is assumed that they can be converted

[1] Short lifetime implies a few cycles such that it can be handled via the processor's pipeline/ forwarding network.

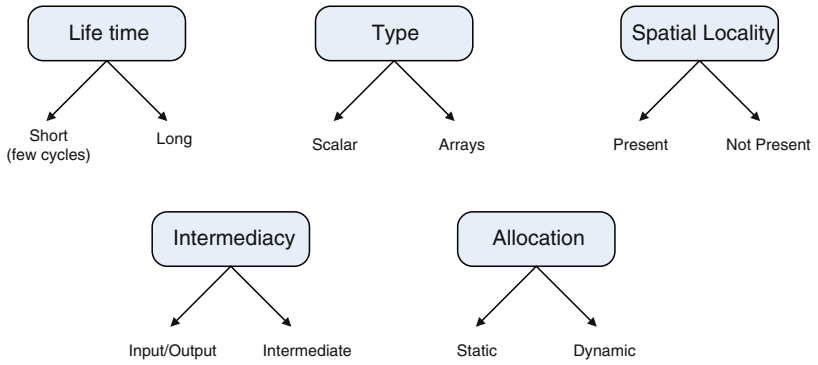


Figure 8.1: Different application variable types

into static variables by the time the compiler has to deal with them. This can be achieved by grouping the data into pools under the control of a dynamic memory allocator [Wil95]. However given the embedded application domain and even in some general purpose domains, there is a set of variables (input/output or intermediate) which exhibit spatial locality. This nature is often not exploited by typical state-of-the-art register file architectures. In contrast there is a small set of variables that exhibit poor spatial locality need to be accessed in random order. This requires a typical register file which can be addressed and written to in irregular order. For an efficient solution each of these set of variables needs to be treated effectively and their properties needs to be exploited. In case the variables do not exhibit spatial locality or their data layout is improper, it is assumed that appropriate data-layout transformations have been done to make the spatial locality is exploitable.

Besides looking at the access part of the variables in application the designer also needs to look into the physical design aspect of the processor architecture. While in most cases during the design phase it may be too early to take into account the physical layout aspect, it is still important that the architecture is defined in a "layout-friendly" way. Given the increasing wire capacitance due to scaling [DeM05, Jos06, Syl99, ITR07], it is important that even in the early design phase the cost of wiring is taken into account. However, even in case interconnect does not scale worse than logic, the proposed register file would still be efficient but the gains of using such a register file may be lower.

This chapter presents a comprehensive technique for organizing the register file such that all the different types of variables are handled in an efficient way. More specifically, this chapter presents one of the main contributions of this book namely a novel asymmetric register file organization called *Very Wide Register* or VWR. This VWR together with its interface to the wide

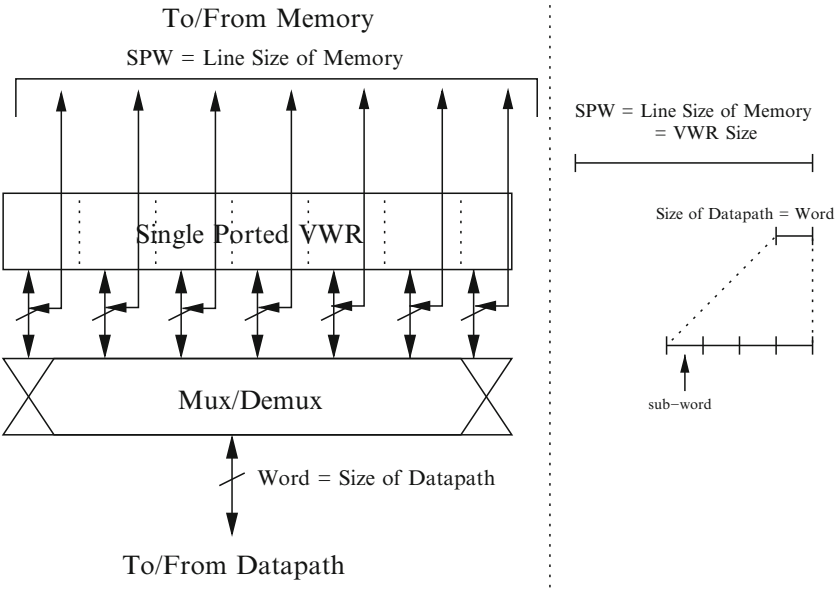


Figure 8.2: Very Wide Register organization

memory, achieves a significantly higher energy efficiency than conventional organizations and forms an efficient and layout-friendly solution for arrays with spatial locality. The proposed register file or foreground memory organization is shown in Figure 8.2. Three aspects are important in the proposed organization: the interface to the memory, single ported cells and the interface to the datapath. The interface of this foreground memory organization is asymmetric: *wide* towards the memory and *narrower* towards the datapath. The wide interface enables to exploit the locality of access of applications through wide loads from the memory to the foreground memories (registers). At the same time the datapath is able to access words of a smaller width for the actual computations (further details in Section 8.3). Internally each of these words can consist of sub-words (for a Single Instruction Multiple Data/ SIMD datapath). The difference between a complete line, word, sub-word is further clarified by Figure 8.2. In this chapter standard state-of-the-art memory is used, with a slightly modified organization compatible with most memory generators: a wide memory organization, where at the interface between the memory and the foreground memory (register file), the complete width of the memory will be read out (complete line/many words).[2] While it is always preferred that complete lines are read from the L1 memory, partial (1/2 or 1/3 section) of the complete line may also be read out and the decode cost of reading multiple words is shared.

[2]A qualitative motivation is given in section 8.3 that for energy efficiency wider memories are suitable.

The rest of this chapter is organized as follows: Section 8.2 motivates the need for a scalable register file architecture for embedded streaming applications from different abstraction levels. Section 8.3 gives a detailed description of the proposed very wide register architecture and its connectivity of the VWR to the memory and the datapath. An example mapping of data to the VWR and its operation is illustrated in Section 8.4. The proposed VWR architecture is compared to the state of the architecture foreground memory architectures in Section 8.5. The experimental setup and the energy savings of using the proposed VWR architecture is presented in Section 8.6. Finally, Section 8.7 concludes this chapter and summarizes the main messages to take from this chapter.

8.2 High level motivation

The motivation for an alternative solution for the register file (alternatively referred to as *foreground memory organization*) of the processor can be brought to attention from various different levels of the design flow.

- **Application:** As explained in the previous section each application has a whole range of variables which are accessed during its execution. Further, the most embedded (especially wireless) applications exhibit a streaming behavior of data (both intermediate as well as input/output variables). Also these variables/arrays exhibit a high amount of spatial locality which is exploited by state-of-the-art techniques in the data memory hierarchy but ignored at present in the register file. Note though that in general the main data types can be indexed in an irregular and partly dynamic way, as explained in the previous chapter. However, even then the techniques proposed there allow to handle this still in the scratchpad memory organisation. Moreover, when copying smaller chunks of data from these larger scratchpads into the foreground memory organisation it is sufficient to deal only with statically allocated and more regularly accessed intermediate variables. These can then be limited to the class of static affine data access. And only that class is further dealt with in this foreground memory oriented chapter.

  On the other hand the data level parallelism of these streaming applications is often limited by dependencies in the application. Therefore a need exists for an architecture which takes both this parallelism as well as spatial locality into account.

- **Processor Architecture:** As explained in detail in Chapter 3, register files form one of the most energy consuming parts of the processor. This is also because of the fact that on average for every operation two operands need to be read from the register file and one operand

written to. Given the high activity it is important that the micro-architecture of the register file is carefully optimized.

Given the strict energy constraints of an embedded processor and the streaming nature of its applications, it is more efficient to program the data movement using SRAM scratchpad memories [Kan04a] instead of caches. In such small L1 SRAM memories (even upto 32KB or 64KB), half of the energy consumption of a read/write operation is spent in the decode logic and another half in the cells [Eva95]. Given the overhead of the decode, it is important that we exploit the spatial locality by loading more than a single word in a single decode stage in the memory. This approach effectively reduces the number of decodes required to get the same amount of data and increases the maximum bandwidth between the memory and the processor simultaneously.

Another key motivation at the architecture level is ports of the register file. Multi-ported register files are known to be energy inefficient and should be avoided if and when possible. Therefore a uni-ported distributed register file solution is needed when possible.

- **Physical Design:** Traditionally in register files, the read/write port of the register file towards the memory and that towards the datapath are identical. In this book this is called symmetrical register file. Due to this symmetrical interface, the wire connectivity between the register file and the memory can become longer. However such a real need to keep the interface towards the memory and that towards the datapath the same does not exist. A customized design of the interface towards the memory can be made which is more efficient and more wire-friendly.

Given the above reasoning about the requirements for an efficient foreground memory solution, this book proposes a new architecture as an alternative for the traditional clustered register file for streaming array variables. The tradition register file is quite capable of covering also the variables with low spatial locality and scalar variables. However, since in embedded systems these variables form the minority, a small traditional register file could also be used to deal with them. A case study of such a split is illustrated in the FEC appendix of [Rag09b] and also the bioimaging ASIP instance explored in Chapter 11 has a need for the scalar variables that are present outside the main Gauss loop nests.

8.3 Proposed micro-architecture of VWR

The architectural motivation for the proposed very wide register architecture is derived from various parts of the processor. Section 8.3.1 gives the architectural innovations in the data memory hierarchy. Section 8.3.2

presents the Very Wide Register and its micro-architecture and also presents the interconnection between the scratchpad memory. The VWR and the interface between the VWR and the datapath is described in Section 8.3.3. The layout aspects of a standard cell based VWR architecture is presented in Section 8.3.4. Finally Section 8.3.5 gives a detailed overview of the micro-architecture of the Very Wide Register in combination with the memory and the datapath if it were to be designed as full custom.

8.3.1 Data (background) memory organization and interface

Energy consumption in memories can be reduced by improving one or more of three aspects: the memory design (circuit level), the mapping of data onto the background memory, the memory organization (and its interface), as discussed in the previous chapter. This section discusses the background (L1 data) memory organization. A detailed energy breakdown of an SRAM based scratchpad[3] shows that for a typical size for the level-1 data memory (e.g. 64KB) about half of the energy is spent in the decoder and the word-line activation [Amr00, Eva95]. The other half is spent in the actual storage cells and in the sense-amplifiers. The decode cost is the price that is paid for being able to access words in any given order. The energy consumption in the memory organization can be optimized further by performing as few decodings as possible by reading out more data for every decode. In the embedded systems domain this can be achieved by aggressively exploiting the available spatial locality of data. While this spatial locality is used for DMA transfers for L2 and L1, cache optimization, etc., it is not further exploited between the transfer for L1 memory and the register file.

In the proposed architecture (see Figure 8.3) spatial locality of data in the L1 memory is exploited to reduce the decoding overhead. The row address (*Row Addr* in Figure 8.3) selects the desired row in the memory through the pre-decoder. The sense-amplifiers and pre-charge lines are only activated for the words that are needed and only these will consume energy and are read out. Figure 8.3 also shows the address organization that has to be provided for such a memory. To be able to handle partial rows (less optimal for energy, but more flexible), the full address contains two additional fields: *Position* decides at which word the read-out will start, while *No. Words* decides the number of words that has to be read out. Hence, at most a complete row and at least one word of the SRAM can be read out and will be transferred

[3] Scratchpad based memories are used in the rest of this book instead of Cache based L1 memories as they have been shown to be energy efficient [Kan04a] as has been motivated in Chapter 3.

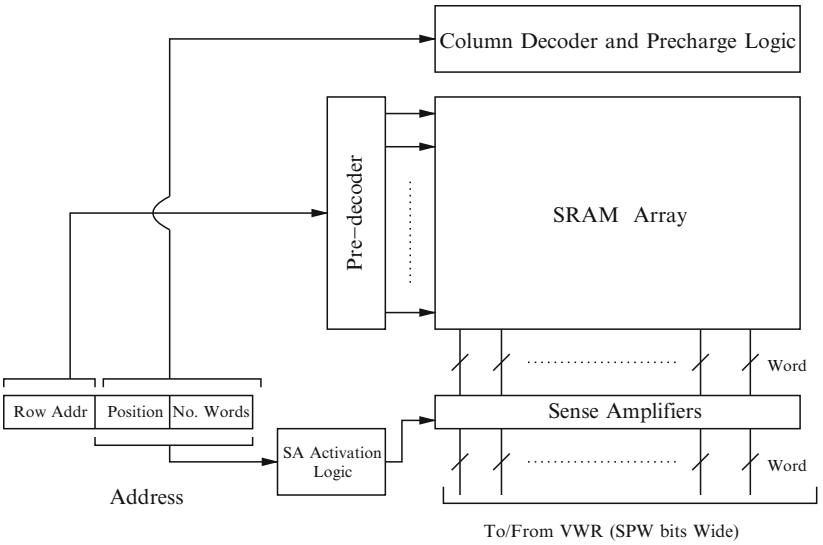


Figure 8.3: VWR and scratch pad organization

from the scratchpad to the VWR registers. The scratchpad can be internally partitioned or banked and the proposed technique can be applied on top of the banked structure.

This architecture is compatible with almost all existing SRAM generators (e.g. Artisan, Virage), but in actual instantiations, such a wide interface may not yet be available. If the used design library does not contain such a wide memory, it can be composed from multiple narrower memories by connecting them in parallel, but the overhead due to extra decoding would not allow the gains to be maximal. For maximal gains it would be necessary that many words (either from the different banks or the same bank) share the same decode circuitry. However, a case study detailed in the FEC appendix of [Rag09b] shows that even with connecting multiple narrow memories in parallel, the gains are still substantial. Concatenating multiple narrower memories still results in gains at the processor level as wider loads/stores implies fewer address computations at run-time, fewer load/store instructions and fewer decodes required even for the register file.

8.3.2 Foreground memory organization

The proposed register file has single ported register cells as shown in Figure 8.2. This register organization is called *Very Wide Register* (VWR). The VWR has asymmetric interfaces: a wide interface towards the memory

and a narrow interface to the datapath. Every VWR is as wide as the line size of the scratch pad or background memory, and *complete* or *partial lines* can be read from the scratchpad into these VWRs. The VWRs have only a post-decode circuit, which consists of a Multiplexer/De-Multiplexer (MUX/ DEMUX). This circuit selects the words that will be read from or written to the VWR. Each VWR has its own MUX and DEMUX, as shown in Figure 8.2. The controls of the MUX and DEMUX on which register is to be accessed is derived from the instructions. Because of the single-ported cell design, the read and write access of the registers to the scratchpad memory and access to the datapath cannot happen in parallel. The VWR is part of the datapath pipeline with a single cycle access similar to register files.

Since the VWR is single ported, it is important that data that are needed in the same cycle/operation are placed in different VWRs. Arrays are mapped on the VWR during a separate mapping process (explained further in Section 8.4). Scalar data like scalar constants, iterators and addresses etc. can be mapped to a separate Scalar Register File (SRF) in order not to pollute the data in the VWR with intermediate results.

The interface of the VWR to the memory is as wide as a complete VWR, which is of the same width as the memory and the bus. Therefore the load/store unit is also different. It is capable of loading or storing complete (or partial)[4]. lines from the scratchpad to the VWRs. Section 8.4, shows an example on how the load/store operations are performed between the memory and the VWR. A more clear example of which data to be loaded/stored and the scheduling and data layout of this data is explained in detail in Chapter 8 of [Rag09b].

To analytically show the gains, assume that M words need to be read and operated on which are stored in the memory. Assume: N words per line in the memory and also N words in one VWR, where $M \geq N$.[5] Conventional register file would require: *M memory pre-decodes + M memory cell activations + M memory post-decodes/column decode + M writes to the register file*, where as in case of the VWR: *M/N memory pre-decodes + M memory cell activations + M memory post-decodes/column decode + M/N wide VWR write*. The ratio between the these two can be of the order of 2–5 excluding the gains in the instruction memory.

Due to the split interfaces of the register file, other optimizations can be exploited at the physical design level. During *placement and routing* the cells of the VWR are aligned with pitch of the sense amplifiers of the memory to reduce the amount of interconnect and the related energy. This enables clear direct routing between the wide memory and the VWR without much interconnect overhead. The same optimization cannot be done in case of a

[4]multiple contiguous words in the same row of the SRAM

[5]Note that this is not a necessary condition, but however gains are higher when M is greater than N.

traditional register file, because of the fact that the memory and datapath interfaces of the register file are shared and due to the multi-ported nature of these register files. For a more detailed experimental setup and analysis of placement and routing, the reader is referred to [Dom05]. A brief overview of the layout aspects in given in Section 8.3.4

8.3.3 Connectivity between VWR and datapath

The foreground memory consisting of VWRs and SRF can be connected to any datapath organization (consisting of multipliers, adders, accumulators, shifters, comparators, branch-unit etc.) by replacing the register file. Figure 8.4 shows the connectivity between the VWRs, SRF and the datapath. The datapath may or may not support sub-word parallelism similar to state-of-the-art processor engines like Altivec, MMX or SSE2.

Once the appropriate data are available in the foreground memory, the decoded instruction steers the read and write operations from and to the foreground memory and the datapath. At a given cycle, one word (consisting of subwords) will be read from the VWR to the datapath and the result will be written back to a VWR. The foreground VWR organization along with the datapath is shown in Figure 8.4.

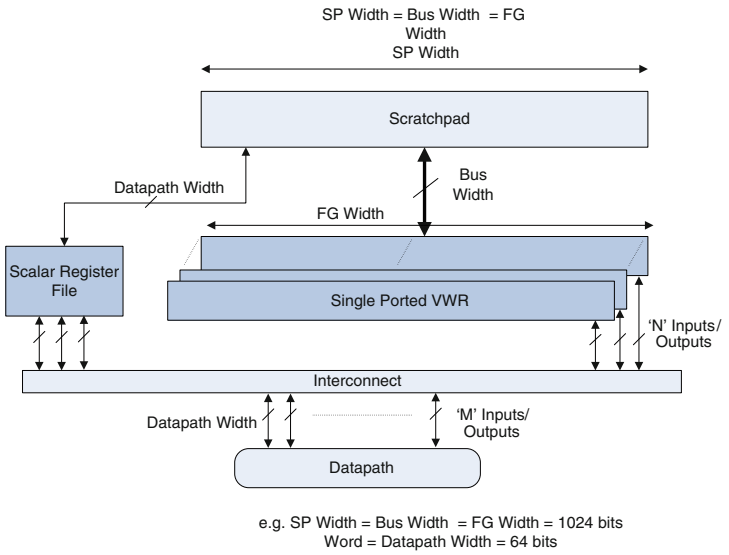


Figure 8.4: VWR and Scalar Register File connectivity to the datapath and the L1 memory

In case the processor is designed with a higher ILP (i.e. multiple instructions can be issued in the same cycle), more VWRs are needed. Given that each VWR is single ported, the number of VWRs needed scales linearly with the number of issue slots. Around three VWRs are needed per issue slot. In case not all issues slots write back to the VWR in the same cycle, the number of VWRs per slot can be lower. This would imply that the VWRs are shared over the different issue slots. More complex schemes can also be imagined where parts of the same word are used as the two operands for an operation. It is expected that the number of VWRs needed would be few as motivated in Chapter 3 that it is more efficient to first exploit the DLP as much as possible and then use ILP to meet the real-time requirements for optimal energy efficiency as well as performance.

8.3.4 Layout aspects of VWR in a standard-cell based design

Given the above micro-architecture it is important that the physical design of the micro-architecture is also done efficiently. Often the design of register files (for small number of ports) is done using register file macro generators. For larger ports register files often full/semi-custom design is used. Since the VWR and its interfaces are unique it is also efficient to use a semi-custom based design methodology.

A tool that can be used for such a design is the DPG or datapath generator [DPG05] from RWTH Aachen. The DPG tool is a tool that takes in leaf cells and places and connects them in a particular order as per the suggestions of the designer. Leaf cells are simple or complex cells (few gates) that can be treated as a unit for the design. The input to the DPG tool is a structural description of the design. In case of the VWR this would consist of the cells and the multiplexers and de-multiplexers. This also gives the freedom per unit on how to shape the unit and place the interface pins of the unit.

For the VWR, the design can be made with such a DPG instance where the leaf cells would include flip flops (for the storage cells) and different muxes/demuxes. Since such a semi-custom design does not have the same restrictions as a standard cell flow where the Vdd and ground lines have to be spaced uniformly over the different rows, a larger freedom exists in the design. An efficient placement can be done such that the pitch of the memory design is aligned with the flip flops of the VWR. The MUX/DEMUX can then be placed below in an efficient way such that the design is optimal.

To further emphasize that a pure standard cell place and route based design would be sub-optimal also for the proposed design, a set of experiments have been performed. A design which consists of a wide memory (960 bits),

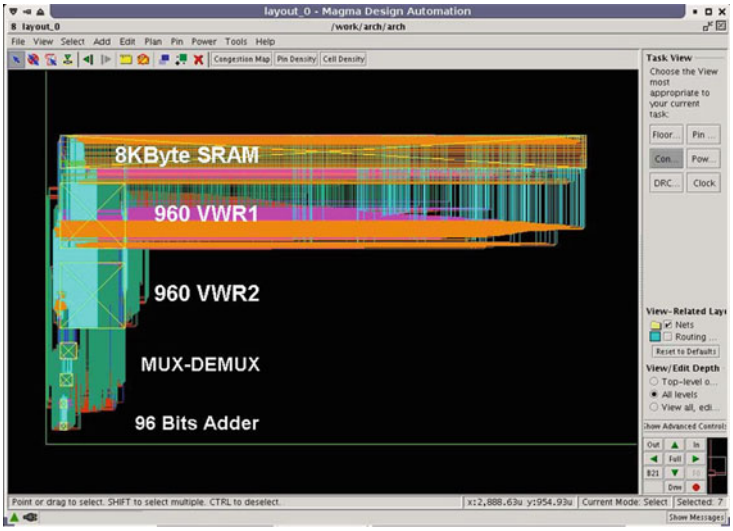


Figure 8.5: Default standard cell place and route

two VWR with 10 entries and an simple96-bit ALU has been used as a driver. The physical design for this has been performed using the following three methods:

1. Pure standard design with default pin placement and default standard cell shape (Figure 8.5).
2. Standard cell design with shaping of blocks and placement but pins automatically placed (Figure 8.6).
3. Standard cell design with shaping of blocks and intelligent placement of pins (Figure 8.7).[6] Such a design can be made using a DPG based flow.

For the design flow Magma Fusion Blast environment has been used. When the individual blocks are optimally shaped and the pins are placed (which is the case with a DPG based flow) an efficient layout can be obtained. This would imply (as shown in Figure 8.7)[7] that the wires would be directly coming from the sense-amplifier (and driver) of the memory directly into the VWR instead of a random routing which is common in pure standard cell design. Figure 8.8 shows the total wire lengths required for the the interconnection between the memory and the two VWRs. It is clear that if a

[6]Note that in this figure the components are kept at a longer distance to show the wiring direction and to reduce the complexity.

[7]Note that in this figure the components are kept at a longer distance to show the wiring direction and to reduce the complexity

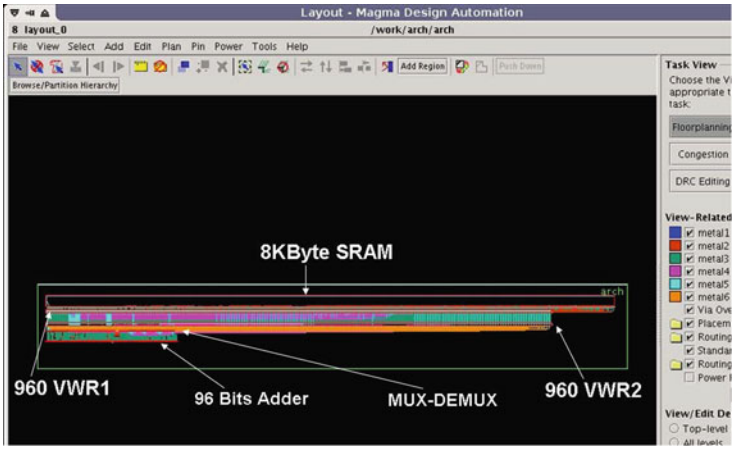


Figure 8.6: Standard placement and routing with optimized shaping of different blocks

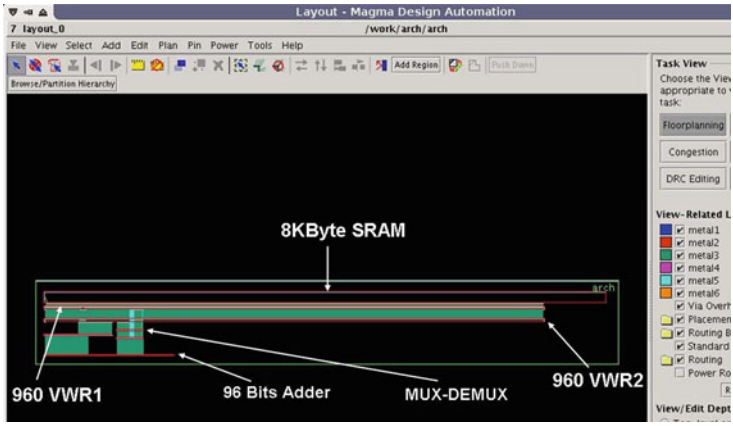


Figure 8.7: Place and route with optimized shaping and pin placement of the blocks

semi-custom design is done, it can be optimized efficiently. For more detailed analysis on the experimental setup used for this and the tool flow the reader is referred to [Dom05]. A basic overview of how the same design could be done in a fully custom design flow is illustrated in the next section.

8.3.5 Custom design circuit/micro-architecture and layout

It was motivated in Chapter 3 that semi-custom design is a viable and necessary design option especially in deep sub-micro technologies for low power.

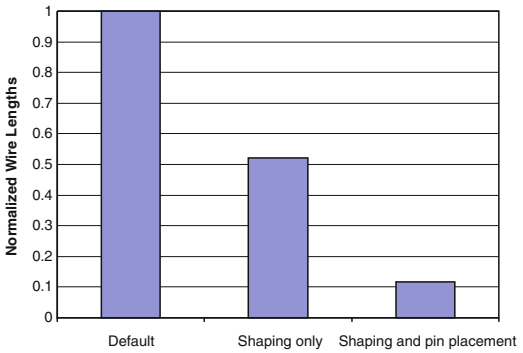


Figure 8.8: Normalized wire lengths for different physical design based flow

This is already often followed in industries with large volumes as well. Also even in case of standard-cell based designs, the register file is at times generated using a module generator. These module generators assembles custom designed parts of the register file. This section illustrates a possible semi-custom design option for a VWR based design. Note however that all the circuit level issues are *not dealt with in detail* and complexity of some circuit level issues is not considered. Therefore the real implementation may vary. Each of the different interfaces and read/write paths are illustrated in the paragraphs below.

All the interfaces are illustrated one by one and for one VWR first. Then the design is generalized for multiple VWRs and how they would be put together. Figure 8.9 shows the one-ported VWR storage cells connected both to the memory as well as to the datapath. The figure shows the interface to the memory which would be used to perform a wide read from the memory and a write to the VWR cells. The cells of the VWR can be pitch aligned to that of the memory cells. A direct connection from the sense-amplifier of the memory to the VWR cells can be made for a write from the memory to the VWR. Partial activation of the sense-amplifiers can be done for a partial write. The read interface to the datapath consists of a network of wires followed by a MUX structure to section from the different words of the VWR. Once the write to the VWR has been done, the wires to the datapath till the MUX are also activated. However these wires can be made low swing. This would ensure that the one-time write cost is low. The sense-amplifier logic can be integrated in the MUX towards the datapath to bring the swing back to full swing. Note that the high-activity wires would be the read to the datapath and the only wires that would switch is the short wires from the MUX to the datapath. The longer wires which include the wires from the VWR cells to the MUX however have low activity. This would ensure the energy consumption is low.

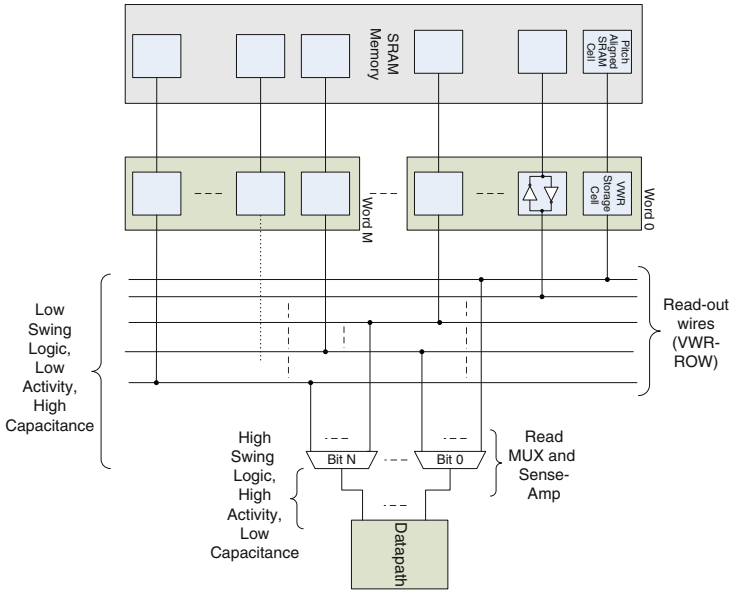


Figure 8.9: Custom designed VWR micro-architecture (write from memory and read to datapath)

The wire routing between the VWR cells and the MUX can be high, therefore efficient routing is necessary based on efficient river routing techniques [Hea91, Lei83] instead of standard-cell based channel based routing technique. While these river routing techniques are from years ago when metal layers were scarce, they can be reused as metal layers have now to be scarcely used due to energy constraints. If the design is fully custom, this can be done efficiently such that each wire does not cross another wire more than once to avoid extra capacitance due to higher metal layers and vias.

Figure 8.10 also shows the VWR with its interfaces to the memory and the datapath. However the figure only shows the interface for writing from the datapath and write back from the VWR to the memory. Note however that a write from the datapath is expensive as a large capacitance has to be driven. To compensate for the large capacitance, a low-swing can be made for the write. The selection of which word in the VWR to write, can be done inside the storage cell with a sense-amplifier. It can be seen from Figures 8.9 and 8.10 that with a single ported cells an efficient read/write can be performed towards the datapath as well as towards the memory. Also longer wires can be made low-swing again to reduce the power consumption further.

Figure 8.11 shows a possible organization when multiple VWRs need to be connected between the memory and the datapath. It can be observed from

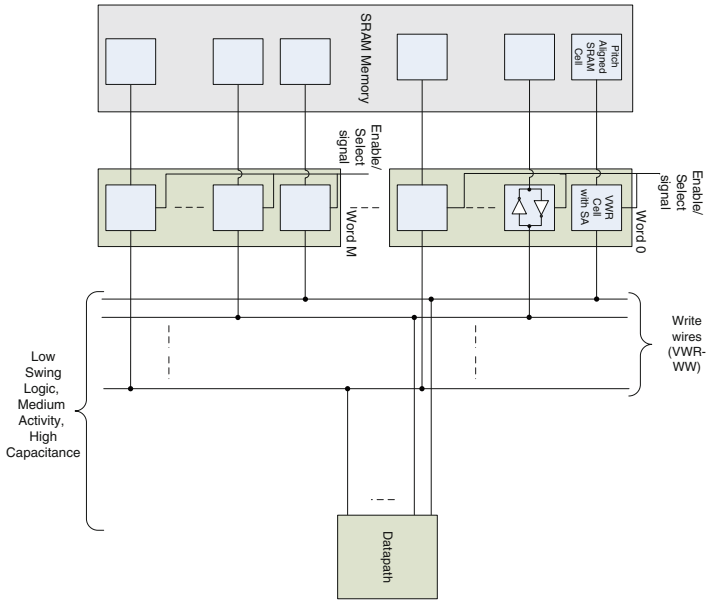


Figure 8.10: Custom designed VWR micro-architecture (write to memory and read from datapath)

the figure that the wire-lengths can increase to a large extent when the number of VWRs increase. This is specifically the case when one SRAM and datapath is involved. However in realistic cases we can expect very few VWRs (two to three) to be required.

8.4 VWR operation

Figure 8.12 presents the operation of the VWR on simplified example code, assuming a 32-bit processor datapath and a 256-bit line-size. This means that one VWR at any given point in time can store eight words. For the sake of simplicity no subword parallelism or vector parallelism is used in this example, of which Figure 8.12 shows the C code (with intrinsics). The asymmetric interface of the VWR results in the following mode of operation: a complete row of the scratchpad is copied to the VWR at once, using a *LOAD_row*. In this example operands from arrays b and c are allocated to two different rows in the scratchpad and to two different VWRs (VWR 1 and 2). Therefore two rows are loaded. In the next phase these operands are consumed one by one by the inner loop and the results are stored in a third VWR (VWR 3). Only when all computations of the inner loop are finished, the complete VWR 3 is stored back to the scratchpad.

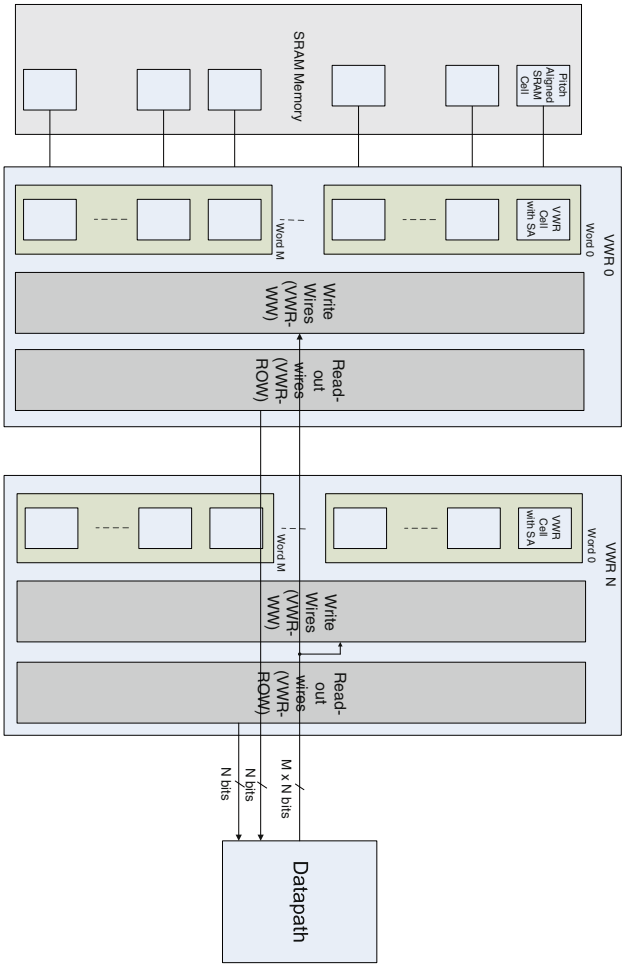


Figure 8.11: Custom designed VWR: Multiple VWR organization

Original Code:

```
for(i=0; i<64; i++) {
  a[i] = b[i] * c[i];
}
```

Modified Code with VWR:

```
for(i=0; i<8; i++) {
  LOAD_row VWR2, b[i*8];
  LOAD_row VWR1, c[i*8];
  for(j=0; j<8; i++) {
    VWR3[j] = VWR2[j] * VWR1[j];
  }
  STORE_row VWR3, a[i*8];
}
```

Figure 8.12: Re-written C code with Very Wide Registers and load/store operations

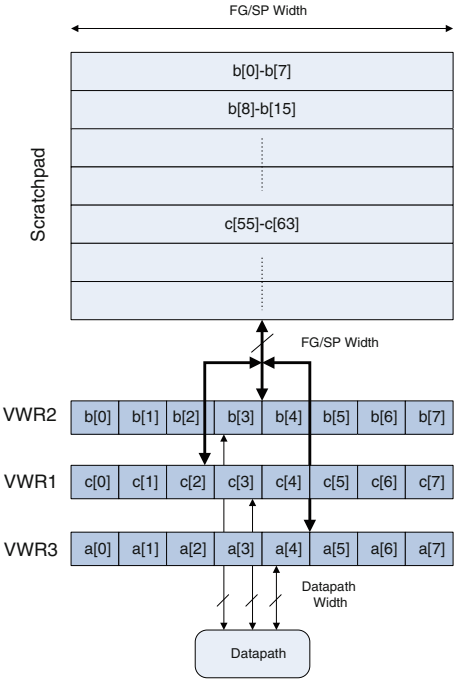


Figure 8.13: Illustration of the data-layout for Figure 8.12 in the different VWR registers and the scratchpad memory

Figure 8.13 shows an illustration of the data allocation in the scratchpad as well as the data layout in the three VWR registers at the end of the first iteration of the outer loop (i loop). The datapath organization can be generic (single/multi-issue etc.). In the first iteration (i = 0), a row of data b[0]–b[7] is loaded onto *VWR* and a row of data c[0]–c[7] is loaded onto *VWR1*. At each iteration of the the inner loop j, one element of b and one element of c from VWRs 2 and 1 respectively are consumed to produced one element of a in VWR3. At the end of the inner loop j, produced data a[0]-a[7] in *VWR3* can be stored back onto the L1 background memory. At the beginning of the next iteration (i = 1), the next set of data (b[8]–b[15] and c[8]–c[15]) can be loaded and consumed in the inner j loop and so on till the end. In this example, there is no need for an epilogue as the number of loop iterations is a multiple of the number of locations in the VWR. If this is not the case, a smaller epilogue loop may be needed for the remaining elements.

Because in the embedded signal processing systems domain (including the benchmarks used here), most data is streaming and continuous in the foreground memory, it is reasonable to assume that most of the time complete

lines of the scratchpad can be loaded with relevant data. It is still possible to load partial rows if not enough independent data words can be found to fill a complete row (for instance at the end of a loop). Also any sort of buffering can be done in the higher level memories such that when the data comes to the L1 data memory and the foreground memory, even non-streaming access can be performed.

Currently the allocation of arrays to the VWR is shown to be done using *intrinsics* (like *LOAD_row*, *STORE_row*, etc.) as the compilation has not been automated. However a technique for compiling C code onto VWR is detailed in Chapter 8 of [Rag09b].

8.5 Comparison to related work

A large amount of research exists on improving the performance and energy efficiency of register file organizations. Many techniques have been proposed at various levels of abstraction (namely circuit, architecture, compiler and system level). In this section we present an overview of the state-of-the-art architectures and emphasize the differences with the proposed organization.

Clustered register file Clustering register files is a generic architectural technique that reduces energy consumption by splitting register files in smaller parts [San01, Lap02]. Since the energy/power consumption increases super-linearly with number of ports, clustering techniques reduce the number ports per cluster and therefore improve energy efficiency. This comes at the extra cost of inter-cluster communication [Rix00a, Lap02]. Fully distributed register file organizations are an extreme form of clustering, where single ported registers are used at the outputs (or inputs) of the functional units [Imp99, Corp98, Jan95]. A bypass network interfaces the functional units in the datapath and the memory. In the proposed approach, multiple VWRs can be viewed as multiple clusters, with at least two or three VWRs per cluster. Each VWR has only a single port. Another category of related work are hierarchical register files [Zal00a] which introduce another level of hierarchy in the register file. While adding an extra level of hierarchy in the register file would exploit the temporal locality available in the application, it would not be able to exploit the spatial locality.

The FEC appendix of [Rag09b] also quantitatively compares and contrasts the proposed very wide register architecture against hierarchical register file [Zal00a], stream register file [Rix00a, Jay04] and traditional clustered register file using the case study example.

Asymmetric register file In state-of-the-art clustered organizations all the ports are symmetric, i.e., the interfaces to datapath and to the memory are of equal width and often shared. In the recently introduced VICTORIA architecture [Der06] from IBM an asymmetrical interface seems to exits, the solution is targeted towards purely a high performance solution with an added constraint of supporting legacy code.[8] IBM's Victoria architecture also requires a run-time lookup table (referred to in their paper [Der06] as map management) which needs to be maintained to get the right set of data from the higher level memory to the register file. This allows the flexibility of shuffling any data element to be placed in any position from the memory to any position in the register file This overhead can be eliminated in case of an embedded system where the application that would be running on the architecture is known in advance, can be statically analyzed and the flexibility is not needed. Each register or the iVMX architecture also requires six read and two write ports which is at the cost of 10 times more area (as stated in the paper itself) and a high power penalty as well.

Register pairs In some processors like the TI's TMS320C64x series [TI00], the processor also allows the usage of register pairs. This allows two adjacent registers to be loaded simultaneously from memory and used either together or separately. However register pairing is limited to two registers, and the register file architecture in [TI00] is still very flexible with no knowledge of data-layout.

Vector register file The concept of wide registers is commonly used in vector registers for data-parallel architectures [Kap03, Asa98, Koz03]. Multiple data elements are stored into the vector registers and one vector/word can be read out to the datapath. This set of data elements is referred to as a word and each data element inside this word is referred to as sub-word. However, in that case the width of the register file and the datapath are equal. The main motivation for wide registers in these architectures is to support data-parallelism. The same operation is performed on multiple data (Single Instruction Multiple Data, SIMD) that are stored in the vector registers.

The proposed approach is complementary to the vector register approach as data level parallelism and spatial locality are orthogonal to each other. Firstly, the primary target is energy efficiency. Secondly, the proposed approach can be used in both SIMD and non-SIMD contexts. In a SIMD context, the widths of data read to the (SIMD) datapath from the VWR are the same as the vector data size. However, a single VWR is much wider than a vector register and holds multiple vector data. Figure 8.2 also illustrates the relative sizes of lines, words and subwords. SIMD datapaths can be used with the proposed

[8] Further the proposed VWR solution has been submitted for patenting at a prior date.

VWR based architecture. In such a case, each line contains a set of vectors/ words and each vector/word contains the different subwords. Usually a large amount of SIMD parallelism is not available in applications due to dependencies and therefore wider datapaths cannot be used. In case of such dependencies, the VWR can still be used, where consecutive words are dependent on each other.

Load/Store Queues Other broad class of related work exists in the area of stall cycle optimization. This work is centered around memory queue structures to reduce the stall cycles. Most of the work [Par03, Set07, Cas05, Sub06] is centered around additional hardware for load/store queues that hide the latency. However such extra overhead comes with the cost of power, which is not acceptable in hand-held embedded systems. Most of these research works are also focused around out-of-order super-scalar processors where the power budget is more relaxed.

DRAM Memory The principle of asymmetrical interfaces have been used extensively in off-chip DRAMs extensively. In DRAMs a complete row of a DRAM bank is latched and the column decoder is used to read words from the latched output. The reason however to do this is driven due to two reasons: limited number of pins to the DRAM chip and the DRAM technology itself. The proposed VWR architecture is partially inspired from the DRAM world. Also in case of the DRAM access the memory controller has to generate the perform the row, column activations.

8.6 Experimental results on DSP benchmarks

8.6.1 Experimental setup

The modeling required for the experiments has been done on the COFFEE framework as explained in Chapter 4. The different components of the processor have also been modeled in the COFFEE framework. The power model used for the different components has been synthesized in *TSMC 65* nm general purpose technology at the worst case corner. The target clock frequency 400 MHz and the power/energy model for each of the individual components have been obtained after place and route and extracted gate level simulation.

Since wide L1 memories and VWR are used here, the energy overhead due to the interconnect bus could become large if the interface bus is incorrectly routed. Hence, the interconnect has been modeled (in both the VLIW and the VWR based register files cases with the L1 SRAM memory) accurately

using the detailed results from place and route and is taken into account in the energy estimations. The energy consumption of the clock tree network of every component is included in the energy consumption of the component itself. Accurate energy/access numbers are obtained from the physical design and are combined with the activation trace for each component from the instruction set simulation, providing the net energy consumption of each component for any simulation run.

8.6.2 Benchmarks and energy savings

For comparison with other regular register file configurations, a 4FU VLIW register file (12 ports, 32 registers deep) is used for the base line processor. Another processor namely a 2FU VLIW (6 ports, 32 registers deep) has also been considered. These two processors have been labeled *4FU VLIW* and *2FU VLIW*. Since the VWRs are single ported, four VWRs are used to allow four (R/W) accesses in parallel. Each VWR contains four words of 32-bit each. A 8KB of memory is used in all the three architectures, but the output is 128-bit for the VWR case and 32-bits for the VLIW. This architecture has been labeled *VWR*.

The TI DSP benchmark suite [TI09d] and realistic benchmark kernels from Software Defined Radio (SDR): 802.11a synchronization, 802.16e synchronization and some benchmarks from Versabench [Rab04] are used to evaluate the three different architectures *4FU VLIW*, *2FU VLIW* and *VWR*. Figure 8.14 shows the energy gains of using a VWR for each of the different benchmarks. Figure 8.14 shows the energy consumption of the data memory, the interconnect between the memory and the register file, and the register file of the different architectures. The energy consumption has been normalized to the energy consumption of the *4FU VLIW* case for each benchmark.

Figure 8.14 shows that the energy consumption is largely dominated by the register file. As expected the *2FU VLIW* consumes lower energy than the *4FU VLIW*. However the *VWR* based architecture has a significantly lower energy consumption compared to both the VLIW architectures. On average the *VWR* based architecture consumes 60% lower energy (considering the memory, buffer, register file subsystem and its connectivity to the FUs) than the *4FU VLIW* case. Another interesting fact to note is that the energy consumption of the interconnect (buffer energy) is also lower in case of the *VWR* based architecture. This is because the pins of the VWR are aligned and matched to the pitch of the sense amplifier of the memories, in order to reduce the net capacitance of the bus and the energy required to drive them. The VWRs consist of cells that are connected to only one bitline and one wordline each, which results in a smaller net-capacitance that has to be driven. Therefore VWR designs can be clocked faster than multi-ported register files. The larger

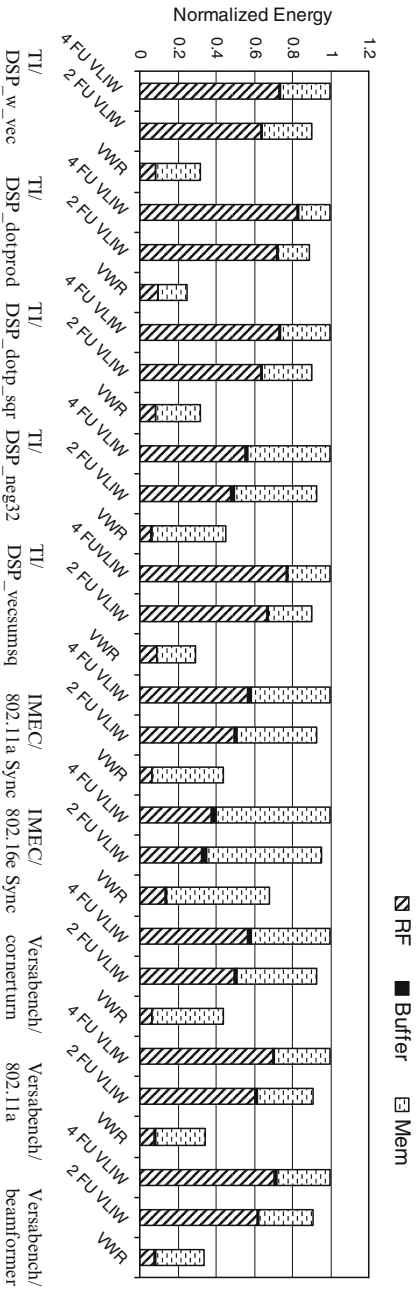


Figure 8.14: Energy consumption split between register file, buffers and register file

buffer cost in case of the VLIW register file is the result of the unified interface and multi-ported nature of a traditional register file to both the datapath and the memory. The extra cost of the bus that connects this port to the memory has to be paid every time the register file is accessed from the datapath.

The gains obtained by using a VWR based architecture differs across the different benchmarks and this is dependent on the spatial and temporal locality exhibited by that application. A more detailed method of how locality can be exploited while mapping data on the VWR is explained in Chapter 8 of [Rag09b].

Note that the gains in the background memory subsystem itself is close to 0. This is because the memory generator used could not generate a 128-bit wide memory and therefore multiple 32-bit wide memories were concatenated and their address lines were tied together. However to obtain additional gains a memory generator is required which can generate a single wide memory.

A more detailed case study and exploration of various processor parameters for an ASIP architecture using the VWR is shown in the FEC appendix of [Rag09b]. It also gives a detailed quantitative comparison between the proposed VWR solution and other state-of-the-art foreground memory architectures. This case study comprehensively explores both the data-path size as well as the memory-interface size of the VWR and other state-of-the-art solutions.

8.7 Conclusion and key messages of this chapter

This chapter has introduced an asymmetrical register file called Very Wide Register as an alternative to the traditional clustered register file for low power embedded applications. Such very wide registers were shown to be efficient for array data with spatial locality. It has been shown that using VWR the designer can exploit the spatial locality provided by many embedded applications in the register file to both improve performance as well as reduce energy consumption. The required formalism to model data in the VWR and a compilation technique for such an architecture starting from C code have been presented in [Rag09b].

CHAPTER 9

Exploiting Word-Width Information During Mapping

Abstract

Optimizing the energy efficiency of an embedded platform has to be tackled at different abstraction levels and for all relevant components. In the previous chapters we have seen that that platform components that initially dominate the power/energy pie chart have been one by one reduced with a substantial factor: the instruction memory organisation, the data background memory hierarchy and the data foreground memory. As a result, the bottleneck in the power pie should now also be strongly influenced toward the last component in the (single-core) platform. Hence, in this chapter we move our focus to the processor datapath. The main objective in this chapter is to use data word-width information in order to reduce the energy consumption of datapath operations of processor-based embedded systems (both traditional instruction-set and coarse grained reconfigurable processors).

9.1 Word-width variation in applications

Figure 9.1 presents an overview of the different aspects of the application mapping that can benefit from this extra information and form the core of this chapter.

F. Catthoor et al., *Ultra-Low Energy Domain-Specific Instruction-Set Processors*,
DOI 10.1007/978-90-481-9528-2_9,

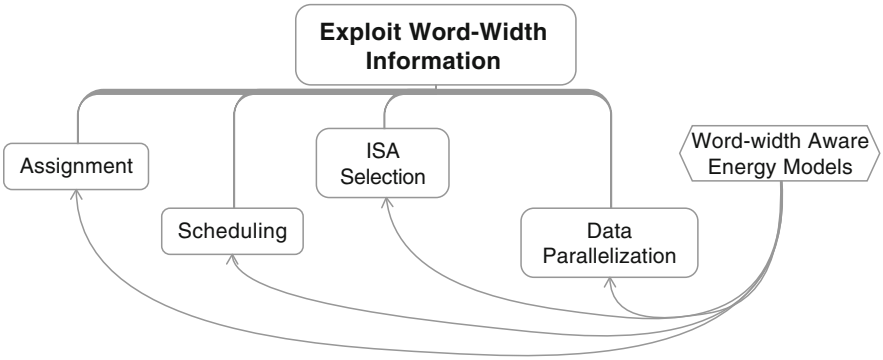


Figure 9.1: Overview of the different ways to exploit data word-width information during application mapping

A representative example will be used to motivate that varying word-widths are to be found in embedded applications and that the optimization freedom that results from this variation is currently for a large part lost due to very coarse rounding after a crude fixed-point refinement (see Section 9.1.2). The fixed point refinement is discussed briefly, as it is an important enabling technique for the work that is presented here. If the word-width information is propagated through different compilation phases, the minimal word-width information for different data can effectively be used during mapping to improve energy efficiency and performance.

As some proposed word-width-aware optimizations have effects on other parts of the platform (e.g. the number of accesses to instruction and data memory is changed because of a change in the number of datapath operations), the impact on the overall platform energy consumption can be larger than what would be expected. This is especially true for systems where the datapath operations account for a rather small relative contribution to the total energy consumption.

The need for word-width aware energy models to improve ISS-based energy estimation sensitivity to word-width variation is investigated (Section 9.2). This extra level of detail is required in order to correctly evaluate and steer optimizations that use word-width information, as is indicated by the arrows of Figure 9.1. Examples of such word-width aware models are presented and the development of these models is described. The improved sensitivity of the energy estimation can influence a designer's decision or prevent wrong conclusions, as is shown for an example energy breakdown in Section 9.4.3.

A systematic description is presented on how to exploit this information during various steps of the mapping process shown in Figure 9.1, namely the

assignment, scheduling, ISA selection and parallelization steps (Section 9.3). For each part, the concept of the optimization is detailed and the expected gains are evaluated.

Section 9.4 will be dedicated to the word-width aware parallelization step called Software SIMD, as the expected gains for this step are more complex to analyze and potential gains are larger than for other word-width aware optimizations.

An efficient mapping of the application code onto the processors of the platform is essential to achieve the best possible energy efficiency. A mapping can be considered to be efficient if it is using the available hardware to its full potential. Exploiting application knowledge during mapping can help to improve the average platform utilization, from the algorithmic level down to the implementation. This chapter presents an approach that enables the usage of word-width information (application knowledge) to evaluate different mapping options in terms of energy consumption and performance.

Word-width information has been exploited during hardware synthesis and the generation of ASICS and application specific custom hardware for a long time. This work specifically targets processor implementations, where still room for improvement exists, as motivated below.

The rest of this chapter is organized as follows. The rest of Section 9.1 discusses the concepts of fixed point refinement and wordwidth variation exploitation. Section 9.2 presents the wordwidth-aware energy models. Section 9.3 discusses three ways to exploit the wordwidth variation, namely scheduling, ISA selection and data parallelisation. Section 9.4 presents the data-parallelisation exploitation based on software SIMD in more detail. Section 9.5 discusses the related work and provides a comparison. Finally, Section 9.6 concludes this chapter and summarizes the key messages of this chapter.

9.1.1 Fixed point refinement

Algorithms are mostly designed using floating point representation. In embedded applications, the floating point data and operations in these algorithms have to be converted to a fixed point format, which is cheaper to implement in hardware (less area, energy and can achieve higher clock speeds). The conversion process is called $fixed-point\ refinement$ or word-length optimization and is summarized here according to [Cma99].

Current fixed point refinement is performed based on the observation that processor implementations can only exploit a very limited set of word-widths. Therefore, current techniques do not perform a very detailed refinement.

This is not a conceptual restriction, however, and the techniques can be used to perform more detailed refinements.

Overflow and precision A fixed-point refinement decides on two aspects of the data: range and precision. This can be interpreted as the required position (with respect to the fractional point) of the Most Significant Bit or MSB and the Least Significant Bit or LSB. Both decisions are different in nature, as the MSB decision aims to avoid errors (overflow is unwanted), while the LSB decision is trying to match the error to the requirements. The distance between the MSB and LSB then fixes the required word-width, as can be seen in Figure 9.2a.

The MSB is chosen such that the largest number that will occur for a certain signal can be represented without causing overflow (based on range analysis). A signal, as shown in Figure 9.4, is a set of data (e.g. an array) that has a certain effect on the application requirements and therefore certain overflow (and accuracy, see below) requirements. A distinction can be made between the $instantaneous\ word{-}width$ at a certain moment in time (one specific value of a signal), the maximal word-width throughout the algorithm execution and over different possible inputs (for all possible values of that signal) and the word-width that is available on the hardware. Different word-widths are defined here as follows:

- Instantaneous Word-Width: Width of a specific value of a signal, at a certain moment in time, during the execution
- Word-Width or WW: Minimal required width of the data at a specific place in the algorithm (irrespective of the hardware width), for all possible values of that signal
- Maximum Word-Width or MWW: The maximum of the word-widths of different signals of an algorithm, such that the MWW does not change throughout the algorithm
- Total Word-Width or TWW: Width as supported by the hardware, width of the datapath

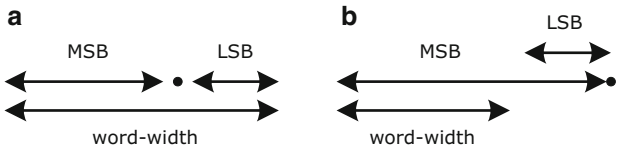


Figure 9.2: MSB and LSB decision for fixed point refinement decide together on word-width

- Dynamic Range or DR: Alternative to the word-widths above, which considers only the varying part of the data in cases where consecutive data are highly correlated (the non-varying part can be removed by scaling the data)

Traditionally, the MWW is taken to decide on the MSB position, after rounding it to the next power of two that is available on the hardware (TWW, worst case). In this chapter, three other approaches are enabled with respect to improving this worst case. Firstly, by taking a more flexible approach with respect to the rounding, the MWW can be used directly without rounding. This corresponds to a smaller worst case width, e.g. 11 bits instead of rounding this up to 16. Secondly, by identifying different phases with different word-width requirements (so-called system scenarios [Pal07, Ghe09]), smaller word-widths can be used for the different parts, which is still the MWW of that phase. In this case, the overhead of switching between scenarios (which then requires a data repacking) should be taken into account when deciding to exploit different phases, which leads to a trade-off between reducing the word-width more and switching more. Thirdly, in an even more complex case, the required width can be separately tracked for every signal by using the WW directly (e.g. for $c = a + b$; *a, b and c* can be of different widths). Depending on the usage of the width information, the word-width to exploit can be chosen from one of the above described cases. To make any of these improvements possible, the lowest possible word-width (per signal, per phase or per algorithm/program) should be decided during fixed point refinement.

The LSB position determines the number of bits that are used to represent the fractional part of the number (Figure 9.2a). The LSB position has an impact on the rounding error (the difference between the floating point number that was used during algorithmic design and the fixed point number for implementation). The highest possible LSB should be chosen such that the application requirements with respect to precision are being met (e.g. a sufficient Signal to Noise Ratio or SNR for audio filtering). The outcome of the LSB determination can be left of the fractional point (as shown in Figure 9.2b), which means that no fractional numbers are needed in order to provide the required precision. For example the LSB on the second position to the left of the fractional point means that only multiples of 4 can be represented. Hereby, a trade-off between implementation cost and data accuracy can be made. Another example of this trade-off can be found in Section 10.3.6, where the accuracy of the operator (a multiplication) is traded off for the cycle and energy cost.

Both MSB and LSB positions can be fixed using analytical techniques (static analysis methods), simulation (dynamic analysis methods) or hybrids. A full description is outside the scope of this text, but the reader is referred to [Cma99, Nov09].

After fixed-point refinement, both LSB and MSB together determine the required minimal word-width. This minimal word-width for every $signal$ can be used as an input to various mapping optimizations. During the rest of this chapter, we assume that detailed information on word-widths is available during different stages of the application mapping. It can still happen that operations of a larger width than the minimally required width are being executed, e.g. due to a fixed word-width in the datapath hardware. However, in that case we assume that the $minimally\ required$ width information is not lost after the initial fixed point refinement, but is still passed on through the different phases of the compiler or is available to the programmer. The fixed point refinement is not the subject of this work, but it is used as an input. More information can be found in [Nov09].

Overflow, underflow and saturation In some cases, choosing the MSB such that it is guaranteed that an overflow (or underflow) will never occur leads to very pessimistic and wide word-widths (e.g. signals with rare transitional behavior) or is even impossible (e.g. digital filters with feedback). In this case, the MSB can be determined with simulation for realistic inputs and saturation behavior can be implemented to catch rare cases. Numbers that are larger than the biggest representable number will be saturated to the biggest value. Extra logic can be added to support saturation behavior.

In this work, it is assumed that no extra saturation logic is used. As the algorithm designer will know at which place in the algorithm the risk for overflow exists, an extra check can be added in software and the required corrective action will be taken (e.g. saturate to maximal value). When exploiting the word-width information, has an effect on the way saturation should be handled, this should be explicitly considered, as is discussed in Section 9.4. In all other cases, the optimizations are orthogonal to the availability of saturation logic.

Scaling of signed numbers For signed data that vary symmetrically around 0, using 2's complement representation, the effectively used word-width will always be equal to the Total Word-Width TWW (datapath hardware width), even if the effective range of values (the true dynamic range) is very small. This is because the sign bit and a large portion of the word flips for a change from a small positive to a small negative number and vice versa. In this work, it is assumed that this undesired behavior can be prevented by scaling the data such that all values become positive, as is shown in Figure 9.3. As the range analysis part of the fixed point refinement already analyzes the minimal and maximal value of each input variable, the scaling offset can be decided during the fixed point refinement. Some precaution is needed for subtraction, but this can be handled and is discussed in Section 9.4.2.

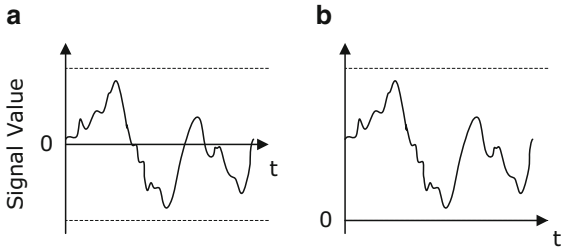


Figure 9.3: Scaling with offset of signed signal (a), to a positive unsigned signal (b)

With respect to complex numbers, it is assumed here that the implementation will decompose complex numbers into their real and imaginary part and operate on them separately (as most embedded processors do not directly support complex arithmetic). Scaling can then be applied to both parts separately.

9.1.2 Word-width variation in applications

Figure 9.4 (from an internal exploration experiment) shows the WW per signal, the result of a fixed point refinement performed for an IEEE 802.11a/g WLAN transceiver for different use-cases, depending on the modulation scheme (e.g. BPSK, Binary Phase-shift Keying etc.) and coding rate that are used. It can clearly be seen that a wide range of widths is available, from 3 to 12 bits per signal, within and across different configurations. Exploiting other design decisions (e.g. trade-off quality of the result against energy spent) and mapping other applications onto the same platform (e.g. audio, video and 3D applications) can lead to a larger diversity in word-widths.

Because of this large diversity, traditional Hardware SIMD support is not cost-efficient (not all widths can be supported), which leaves room for other techniques to exploit this further. Even though the fixed point refinement can determine the exact minimal width, this information is currently not used in processor implementations (it is for a part used in ASICs, which contributes to the significantly better energy efficiency of this type of fixed hardware). Because processor datapath hardware only supports operations of a single width or a limited set of fixed widths (e.g. 32-bit datapath with additional support for 2×16 or 4×8-bit SIMD), the minimal width is rounded to the next available power of 2, e.g. minimally 6 bits is rounded to 8 bits. Additionally, because the hardware does not support all the different widths, this information is not exposed to the compiler (or made available to the programmer), and therefore no further optimizations (even if they would be

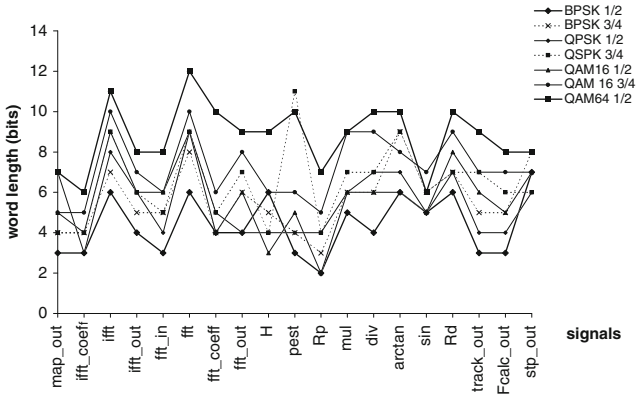


Figure 9.4: Minimum required word-lengths for different baseband configurations (modulation scheme and coding rate) of an IEEE 802.11a/g WLAN transceiver

supported by hardware of a fixed width) can exploit this diversity. In conclusion, the standard practice to round different widths in applications to a common maximum and therefore worst-case width can still be improved. To enable this, the minimal width should be propagated to the mapping (compiler or programmer).

To better exploit the large word-length diversity, this chapter presents a description of different mapping techniques, allowing designers or compilers to exploit the $minimal$ and $heterogeneous$ word-width information in ways that are not currently exploited. The primary goal is to explore how this information can be exploited and to estimate the potential gains in the context of embedded applications that require extreme energy efficiency and a high performance.

9.2 Word-width aware energy models

To enable designers to see the potential gains of using word-width information during mapping and achieve a better energy efficiency and higher performance, simulation and estimation techniques need to give more accurate (specifically more sensitive to word-width variation) and more detailed energy estimates. To make this possible, word-width aware energy models are needed. Current energy models used in ISS energy estimation assume that hardware components (e.g. adders, multipliers) are always operating on data that fill the complete width of these components. When the actual required data width of a certain algorithm is less wide, these components

internally toggle less, which leads to a smaller energy consumption. Current processor and platform simulators can easily be extended to make use of these more detailed energy models, once they are available and if the word-width information is propagated or can be extracted.

A complete description of the used modeling approach is presented in the appendices of [Lam09]. This section will briefly summarize the presented approach, show an example of a word-width aware model and briefly discuss the use-cases, benefits and the usage, with respect to the standard, activation-based models.

9.2.1 Varying word-width or dynamic range

To model variations in word-width or dynamic range, the input data that are simulated in the energy estimation flow, have to exhibit this variation. A set of testbenches have been created that contain automatically generated input data, containing a random component, but with the following statistical characteristics:

- Word-Width or WW: For a component that has a Total Word-Width (TWW) of e.g. 32 bits, only the WW least significant bits are assumed to be toggling, while the other bits are always 0.[1]
- Dynamic Range or DR: In this alternative, only DR bits out of the TWW are toggling, but the other bits are fixed, but not all zero.

For both cases, we assume an initial distribution of 50% 0 bites and 50% 1 bits. The data has been generated such that it has an activity of 0.1, 0.2, 0.3, 0.4 or 0.5. The activity is defined as the probability that a bit (of the *active* part of the word) toggles between two consecutive data words.

An example of such a word-width aware energy model for a simple Carry Lookahead Adder can be seen in Figure 9.5, where the activity is indicated next to the used testbench type WW or DR. Similar models have been generated for other processor components, like a multiplier and a register file.

In contrast to traditional non-word-width-aware models that only provide an energy per activation for the full word-width, which corresponds to the right-most points in the graph, the presented model includes energy per activation estimates for various word-widths, including a variation in activity.

[1] In this work, it is assumed that all data has been scaled to be positive, in order to prevent the complete datapath width to toggle for data varying around 0 when using 2's complement representation.

9.2.2 Use-cases for word-width aware energy models

It is needed to go through the effort of generating these models at least once per technology, in order to verify if a sufficient variation in energy consumption exists to motivate the extra level of detail. If a sufficient variation is found, then extra optimizations that exploit word-width information can be enabled and e.g. mapping decisions can be based on word-width aware energy estimates. The experiment that is presented in Figure 9.5 shows that the energy per activation of e.g. a 40-bit addition is about 30% less than for a 64-bit addition on the same adder (e.g. for an activity of 0.5, and varying word-width WW). Therefore, we conclude that for a 90 nm technology, the extra effort is justified if the target application uses a data word-width or dynamic range that significantly differs from the hardware word-width.

The word-width aware models can be seen as a replacement of the activation based models that are currently being used with ISS-based energy estimation that is specifically more sensitive to the word-width variation and in that respect more accurate for relative comparisons. As such, they can give designers fast and fairly accurate energy estimates to steer the energy-aware mapping of an application onto a platform. The effort that is needed to move to word-width aware energy estimation differs according to the goal of the experiments at hand.

For processor design, which includes architecture exploration, it is the goal to optimize the architecture components given a set of applications or an application domain. Here, models should be developed for every component

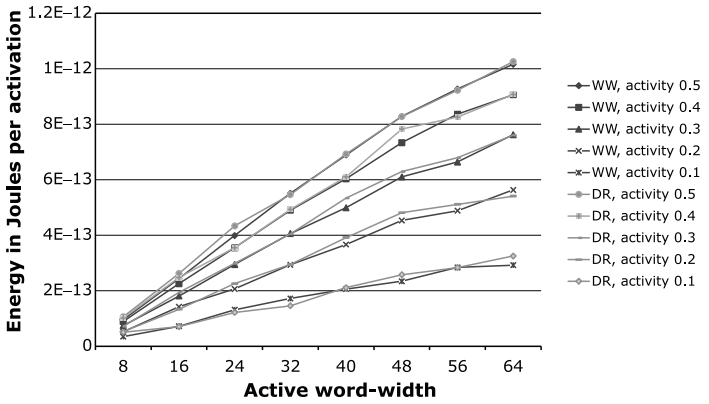


Figure 9.5: Word-width aware energy model for 64-bit Carry Lookahead SIMD Adder in 90 nm CMOS standard cell technology. WW and DR indicate the word-width or dynamic range respectively, for different bit activities ranging from 0.1 to 0.5

within the explored range (e.g. adders of different widths and potentially of different types, but also for register files of different width, depth and for different numbers of ports, etc.). This quickly leads to a high modeling effort. This modeling should only be done once per technology node, making it still very worthwhile. Section 9.2.3 presents an estimate for one such architecture based on energy models generated for a single adder, multiplier and register file.

Mapping experiments, where the architecture of the platform is given, require only the word-width aware energy models for specific instances of components present in the processor. This means e.g. only one type of adder of a fixed width needs to be modeled, e.g. the Carry Lookahead Adder model shown in Figure 9.5. This heavily restricts the modeling effort that is required, making it a feasible and practical approach.

In both cases, the energy models are used to correctly assess the bottlenecks that would decide which part to focus on next and potentially to steer word-width aware optimizations (as are discussed in Section 9.3).

9.2.3 Example of word-width aware energy estimation

A representative wireless communication kernel has been mapped onto a fixed platform, containing a small in order RISC processor and both a data (DL1) and instruction (IL1) memory. The processor has a 64-bit datapath that consists of the components that have been modeled above. The data is read from and stored back to a background data memory (DL1) of 8kB and instructions are read directly from the instruction cache (IL1) of 8kB.

The result presented in this section are for a kernel from a real life MIMO (Multiple Input Multiple Output) Baseband processing algorithm, namely the Spatial Equalizer part.

Figure 9.6 shows rather extreme comparison to make the point, comparing the *Activation Based*, as has been discussed in Chapter 4 of [Lam09], (No WW) approach for 9-bit data on a 64-bit datapath counted at the full 64-bit cost with the *Word-Width Aware* (WW Aware) for 9-bit data on a 64-bit datapath, but counted using the word-width aware model of 9-bit data on a 64-bit datapath.

The energy breakdown for the Activation Based method shows a severe overestimation of the energy consumption. The Word-Width Aware method (WW-Aware) more accurately estimates the real cost, when only a small width of the available datapath is used. For the datapath this effect is most visible (a factor 3), as a reduced active word-length heavily reduces the activity in the datapath units, like adders, shifters and multipliers.

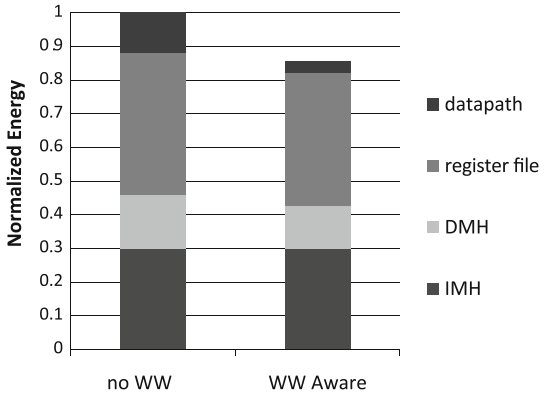


Figure 9.6: Comparison between activation based energy estimation (*no WW*) and energy estimation based on activation information using word-width aware models (*WW Aware*) for the MIMO Spatial Equalizer

Figure 9.6 shows an overall difference of 14.4% in the energy estimation of the kernel for the full platform, between the non-word-width aware and the word-width aware method. This is a quite significant difference and motivates the development of word-width aware models in cases where the datapath operations are really the bottleneck and there is often a large mismatch between the datapath width and the effective width of the data. Not doing so, can in these cases potentially lead to wrong conclusions, or wasted effort in optimizing the platform. Based on the *no WW* plot, one might e.g. decide to spend effort on techniques that try to reduce the energy spent in the datapath, even at the cost of a small overhead. In reality the cost of the datapath is much lower and that small overhead could introduce an overall energy penalty. It should also be noted that the exact difference depends on the chosen processor and platform.

For more information on the modeling in this experiment and an analysis of the impact of word-width variation for the different components, the reader is referred to the appendices of [Lam09].

9.3 Exploiting word-width variation in mapping

This section describes how transformations or mapping techniques can exploit knowledge about data word-widths or a variation there-of in order to achieve a better energy efficiency or performance. The focus of this work is to exploit word-width information during various stages of mapping operations onto the datapath. The secondary effects on other parts of the platform are

taken into account, e.g. when the number of accesses to register files, data or instruction memories is changed. Specific techniques to exploit word-width information in other parts of the platform, e.g. to reduce the required communication bandwidth between processors or to reduce the memory footprint of arrays, are outside the scope of this work.

The rest of this section describes the use of word-width information during four phases of mapping operations to the datapath, namely during assignment, scheduling, ISA selection and parallelization. The optimization concepts and initial estimates on gains that could be achieved during these steps are presented. As was motivated in Section 9.1, it is assumed that the word-width information is available during mapping and that it represents the minimal word-width that will respect the application requirements.

9.3.1 Assignment

A first way to exploit detailed word-width information, is during the assignment of operations to PEs and FUs in these PEs. The assignment step decides onto which FU a certain operation is mapped, taking into account the availability and the type of operation that needs to be executed.

9.3.1.1 Concept

It is possible to design a processor with multiple different implementations of the same type of FU in the same PE. One example can be found in the separation of a fully flexible shifter into a shuffler/shifter combination [Rag06b]. In this case, a flexible shuffle network can re-order or move, e.g. 4-bit parts of a word efficiently, while the complexity of the coupled shifter is heavily reduced and only needs to support shifts up to a 3-bit distance. Correspondingly, separate implementations for often occurring specific word-widths can be added. This would make sense if they would represent different trade-off points, e.g. a fast one with restricted functionality and low energy consumption vs. a slow, flexible and more energy-consuming one. As such, the functionality restriction can refer to a maximum word-width that is supported and the assignment can be based on the word-width information.

Figure 9.7 shows a schematic data-parallel datapath, in which multiple implementations are provided for the multiplier and the shifter. For a certain application domain, a large number of multiplications could be with very small factors of a restricted width. A cheaper implementation of this restricted multiplication could be provided, next to a more costly wider multiplier. Related to this optimization, the implementation of constant multiplications and trading off multiplier accuracy with the operator cost is explored in Chapter 10.

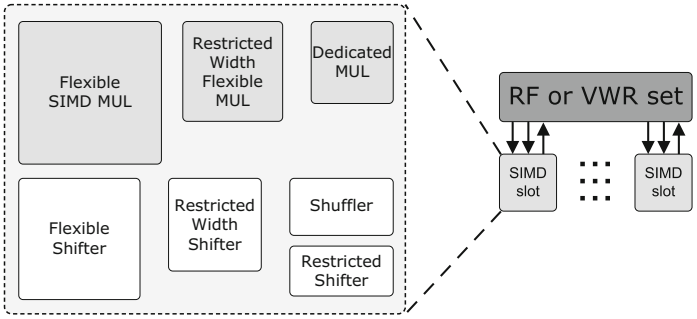


Figure 9.7: Word-width aware assignment optimization on a schematic SIMD datapath

As was proposed in Section 3.6.6, splitting address computations and data computations onto different PEs altogether would be one example of word-width aware assignment. Another possibility would be to provide different FUs customized to the address and to the data widths inside every PE and to make the split internally.

9.3.1.2 Expected gains

The expected gains of the word-width aware assignment depend heavily on the target application, on the available word-widths and the ratio between different widths and on the energy versus performance trade-off points offered by the different FUs. However, it is already clear that the decision to which exact FU inside a fixed PE a certain operation will be assigned, will have no other effect on the number of accesses to the register files or the instruction memory. Therefore, the cost improvement would be restricted to the relative contribution of the datapath operations to the total cost. Only when different FUs have different latencies in terms of cycles, the impact on the schedule length could indirectly affect the execution cost beyond the datapath logic alone. This effect will be discussed further, specifically for different FUs to implements multiplications, in Chapter 10.

Scope, problems, benefits etc. In order to be able to exploit the assignment optimization, different assignment options are required, which means multiple resources should implement the same functionality. This would lead to a hardware overhead and extra control requirements (extra instruction memory cost). It is unlikely that extra resources would be added specifically to be able to use this optimization. However, in some cases, this choice will be available for other reasons. Then, word-width information can be used to exploit that freedom or improve the assignment decision.

As was mentioned above, the datapath/address path split is based on one specific instance of word-width aware assignment. However, it also enables a better data-layout for, e.g. VWR usage (see Section 3.6.3). Therefore, this split is still worth-while, even if the datapath logic only consumes a small part of the platform energy.

In conclusion, we do not expect word-width aware assignment (apart from the split of datapath and address path) to be a general purpose optimization, but it can be part of the optimization toolbox in certain specific cases, e.g. if multiple FUs are already available for other reasons, like with the goal to provide a performance range or to provide redundancy for unreliable technologies. Therefore, since our target application domain is currently not dominated by the datapath operations alone and as we have already proposed a split of data and address operations, this optimization is not explored further in this work.

9.3.2 Scheduling

During scheduling, the order in which the operations are mapped onto the datapath is decided (by the programmer or by the compiler). In this step, the dependencies between the operations and the latency of each operation has to be respected. If word-width information on the data that are consumed by these operations is added to the set of information that can be used to make the scheduling decision, a more energy efficient schedule can be constructed as is described below. This is a second potential usage of word-width information in the compiler.

9.3.2.1 Concept

When consecutive operations on a certain functional unit (e.g. an adder) operate on data of the same word-width, which is smaller than the total width of that unit, less energy will be spent because not all circuitry is activated (as was shown in Figure 9.5). When operations on data of different widths (e.g. as is shown in Figure 9.8a) are being executed successively, the active part is the maximum word-width of both (indicated with striped boxes). If this maximum width is less than the total width of the datapath, using the corresponding value from the word-width aware energy model will then be more accurate.

When the application contains different groups of operations, e.g. filter operations on one side and updates on filter coefficients on the other side, which have different accuracy requirements and are operating on different word-widths, there is a possibility of doing a word-width aware scheduling.

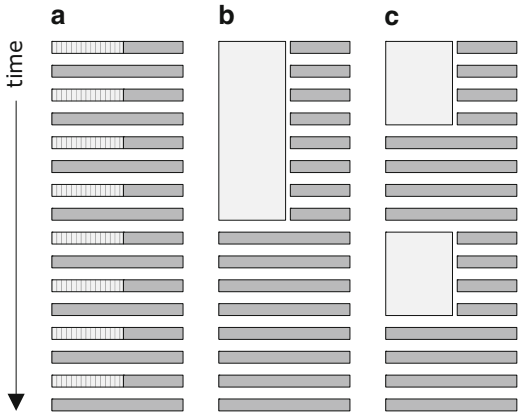


Figure 9.8: Schematic representation of word-width aware scheduling optimization, in which the order of the operations is modified to minimize toggling in the datapath logic

By grouping the operations that operate on the same word-width, the internal activity of the hardware can be minimized. Figure 9.8b shows a grouping of the operations of equal width, which leads to less toggling, indicated by the large rectangle. Dependencies between the operations have to be respected, which in some cases results in the successive execution of smaller groups (as shown in Figure 9.8c).

9.3.2.2 Expected gains

The expected gains for this optimization can be directly derived from the energy models for the datapath FUs, as presented in Section 9.2. They depend only on the total width of the FU and on the two different widths of the effective operations.

Figure 9.9 presents an initial estimate on the gains that could be expected, using a hypothetical example: a series of 100 additions on two data sets of different widths (assuming an equal number of operations on each width). Estimates are presented for different dependency distances between the operations and for different variations in word-widths. The original cost, shown on the left, assumes subsequent operations on different widths of respectively 12 bits followed by 8 bits, 32 bits followed by 12 bits and an extreme case of 32 bits followed by 4 bits. All following bars are normalized with respect to the base case and present a dependency distance of 2 (additions of same width grouped per 2) up to the best case (all additions of the same width grouped).

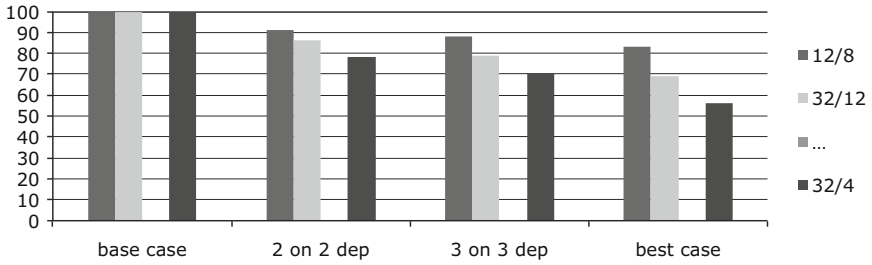


Figure 9.9: Example of energy savings as a result of word-width aware scheduling optimization

From this example, it can be seen that if very heterogeneous word-widths are present and if no close dependencies between these computations exist that prevent a re-scheduling, the potential of word-width aware scheduling can reduce the energy consumption of the datapath logic with 10–45% (e.g. with 20% for the 32/12 case with a 3 on 3 dependency in Figure 9.9).

Scope, problems, benefits etc. As this optimization only changes the order of the operations (respecting the dependencies), it has no effect on the number of accesses to data or instruction memories. Therefore, the potential is restricted to reducing the energy cost of the datapath operations. This corresponds to only 1.2% of the full platform cost of Figure 3.5, but this can increase for more parallel architectures as is shown in Figure 3.9 (12% of the CGRA processor: 5% for the ALU PEs and 7% for the MUL PEs). As the base cost by far exceeds the variable cost for the pipeline registers, this reduction is minimal and is not shown here. The same applies for reads and writes to the register files (as shown in Figures 9.10 and 9.11), so it does not make sense to attempt a toggle minimization for subsequent accesses to the register files (this however, can change if better register file designs would reduce the base cost).

However, this optimization can be applied within the restrictions of other scheduling constraints, at no extra cost. Therefore, it can be applied for application domains where heterogeneous word-widths are available, the datapath operations are a bottleneck and the last Joule needs to be optimized (e.g. biomedical signal processing). Since this is clearly not always the case, this is not a general purpose optimization.

A clear link exists between this optimization and the proposed split of the data and address computations as was discussed in Section 3.6.6. The aim of word-width aware scheduling is to reduce the amount of toggling by grouping operations on shorter word-widths instead of mixing them with operations on longer word-widths, which would increase the effectively toggling width to the width of the latter. One of the main reasons to propose the

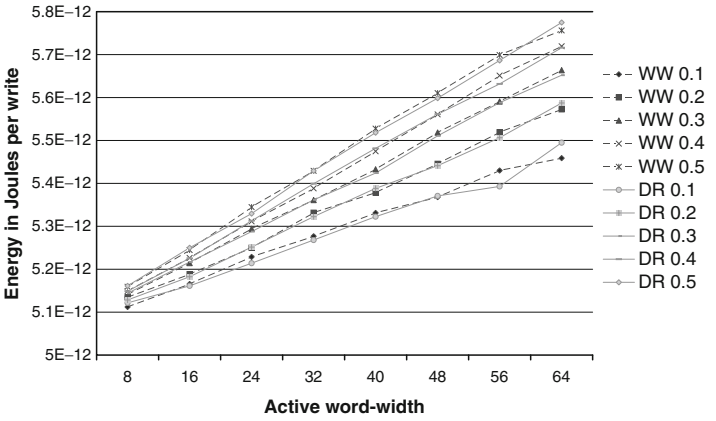


Figure 9.10: Energy per write for a 1W/2R Synthesized Register File that can store 16 words of 64 bits

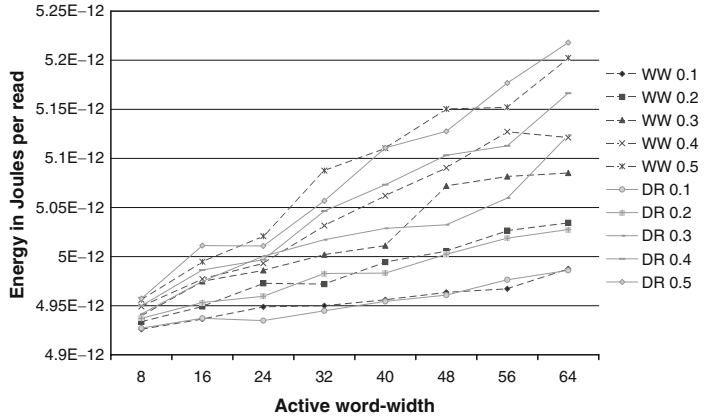


Figure 9.11: Energy per read for a 1W/2R Synthesized Register File that can store 16 words of 64 bits

split of data and address operations onto different execution units altogether (which is an assignment decision, as is discussed in the next section), was the difference in dynamic range between both types of computations. This difference in dynamic range is therefore one of the most obvious cases, in which also word-width aware scheduling can lead to a gain. However, when the operations are split onto different execution units, the potential of word-width aware scheduling is reduced, as data and address operations are already effectively split.

Word-width aware scheduling hinges on the assumption that some operations under-utilize the full width of the datapath and that grouping those

operations can still minimize the toggling on that part. Therefore, the potential gains of this optimization are reduced by other techniques that attempt to fill the available hardware as much as possible (e.g. SIMD, as discussed in Section 9.4).

As with the word-width aware assignment, our target application domain is currently not dominated by the datapath operations alone and as we have already proposed a split of data and address operations, this optimization is not explored further in this work.

9.3.3 ISA selection

Another way to exploit word-width information, is when the intermediate representation or the Data Flow Graph (DFG) of the application is parsed and virtual operations are mapped to instructions of the target processor. In VLIW terminology, the combination of all operations of the different slots is called a single instruction. To prevent confusion, the ISA (Instruction Set Architecture) Selection can be called operation selection here. In some cases, the data word-width can provide extra information on which mapping strategy to follow.

9.3.3.1 Concept

In a traditional compiler, the mapping of DFG nodes to operations of the target processor is decided based on which set would cover the DFG in the most efficient combination of operations (mostly with respect to performance). In some cases, the additional word-width information can influence this decision.

In Chapters 10 and 11, two examples of word-width-aware ISA Selection are described. There word-width information, together with accuracy requirements and the specific type of multiplication at hand are taken into account when deciding on how to implement a multiplication (see Chapter 10 for the specific details). Figure 9.12 presents a graphical illustration of how the multiplication node in the DFG can either be directly mapped to a multiplier (for wider word-widths and high accuracy requirements) or can be broken down into a set of cheaper add, shift or special new operations if the word-width (and thereby the number of required instructions) or the accuracy is less.

9.3.3.2 Expected gains

ISA selection can in general be described either as a one-to-one mapping (one node in the DFG corresponds to a single instruction), a one-to-many

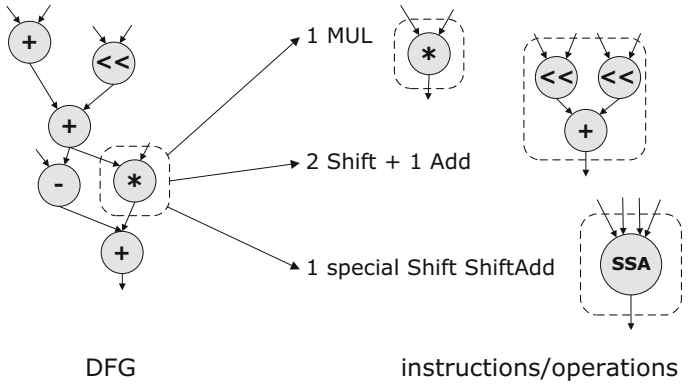


Figure 9.12: Schematic representation of converting a DFG node into one or a set of instructions/operations on the target processor

mapping (one node is broken into multiple operations), a many-to-one mapping (many nodes are combined into a single operation, e.g. using template matching techniques) or a many-to-many mapping (a combination of nodes is replaced by a combination of other nodes). Because of this variation in resulting operations, a different number of intermediate variables will have to be register allocated and the resulting schedules can vary in length. Therefore, the potential effect of this optimization goes beyond the datapath logic alone, as the number of accesses to register files and the Instruction/ Configuration Memory Organization (ICMO) can change. Consequently, the expected gains are difficult to predict in general and are more complex to estimate than for word-width aware scheduling and assignment. The specific case of strength reduction for constant multiplications, as covered in Chapter 10, will thoroughly cover one example of this optimization.

9.3.4 Data parallelization

One more potential step during application mapping that can be improved when using the additional word-width information is when deciding how much Data Level Parallelism or DLP to exploit. This decision depends on the amount of parallel independent operations that can be extracted from the program (e.g. across loops as described in Section 3.6.5), but in practice the choice is restricted to what is supported by the hardware: how many operations of a certain width can be executed in parallel.

The same hardware-supported widths are traditionally used as the minimal word-widths after fixed point refinement. By removing the restriction that all

used data widths should be equal to what is supported by the hardware and by propagating the real minimal word-widths, more optimal mappings can be constructed.

9.3.4.1 Concept

When parallelizing the execution of a kernel (using SIMD), the same operation is executed for multiple data of equal width in parallel. This width is chosen from the set of widths supported by the SIMD hardware and is mostly equal to one of the following powers of 2: 4, 8, 16 or 32.

Potentially, a different, more efficient parallelization can be achieved for data of which the minimal word-width differs from the available hardware options. Variable width hardware has been proposed, for which multiple custom word-widths are supported (e.g. for 9-, 18-, 24- and 36-bit words in the Mpact Media processor [Fol96]), but providing a per bit variable SIMD support in hardware is prohibitively expensive. Other solutions use a bit-serial approach to offer a range of flexible widths, but they do not provide the required performance [Ann90]. Above all, they are very energy inefficient, due to the overhead in control and storage. As a fully flexible Hardware SIMD solution is either too expensive (area, energy) or too slow, an alternative approach is proposed here.

Using detailed word-width information, different data words can be packed into a single word in order to operate on them together, without specific hardware support for the combined word-widths. Without specific hardware support, however, keeping the data separated during the execution is no longer guaranteed. By inserting extra masking operations where needed and potentially extra guard bits between the data when packing them together into a larger word, the correctness can still be guaranteed. This technique is called Software SIMD.

Since Software SIMD is less restrictive toward the word-widths that can be combined, it can handle cases where traditional Hardware SIMD is not possible. When compared to the traditional $hardware$ SIMD, the potential lies in operating on different word-widths together, called $Heterogeneous\ Software{-}SIMD$, or operating on widths that are not equal to the supported hardware widths or a combination of both.

In the following simple motivating example (Figure 9.13), minimal word-width information is used when deciding how to make use of SIMD to handle data parallelism. The minimal width of the data that is needed to reach the required precision is decided upfront. It is then passed on to the designer or to the compiler and can be used to decide the optimal packing of words into the full datapath width.

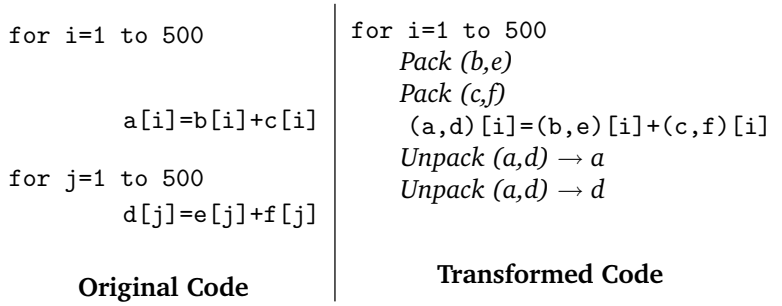


Figure 9.13: Motivating example for Software SIMD, in which the word-width information of a, b, c, d, e and f is used to pack them together into a single word, even if this would not be possible using traditional SIMD

The original code on the left shows two loops that are operating on arrays. During fixed point refinement, the data of arrays a, b and c are found to require minimally 18 bits, while the data of arrays d, e and f only require a 12-bit precision. On a traditional 32-bit datapath, supporting 4 × 8 or 2 × 16-bit SIMD, the loops can not be merged by using SIMD. Because of register file pressure, we assume both loops cannot be merged.

In the right code fragment, the extra word-width information was used during mapping, and Software SIMD is used. By packing data of 18 bits with other data of 12 bits into one 32-bit word, leaving a 1-bit guard band at the MSB side of each sub-word (not always needed, as explained in detail in Section 9.4), SIMD can still be used. Hardware SIMD support is not required and the datapath is still filled. In this case, the computations of both loops have been merged and arrays `(b,e)` and `(c,f)` are operated on together, producing the combined results `(a,d)`. In this simple example, it is assumed that register file pressure is reduced by combining data into less words, so the loops can be merged.

9.3.4.2 Expected gains

Parallelization by using SIMD is a very popular technique, as it increases performance and improves the energy efficiency. In contrast to the above discussed word-width aware scheduling and assignment optimizations, changing the parallelism that is exploited by using a more flexible Software SIMD does change the number of operations and also the accesses to the register files and the instruction memory. Therefore, the impact of this optimization goes beyond the cost of the datapath operations alone. Because it affects the number of accesses to memories and also the number of cycles needed to complete an algorithm, the gains depend on different parts of the platform (not only the 1.2% of the datapath operations of Figure 3.5, but all parts of

the core, which is in total over 60% and even the accesses to background memory could be affected). If the technique is applied blindly, the overhead can be larger than the gains.

To be able to operate on the data together, without the use of hardware for that specific SIMD width, the sub-words have to be packed in a certain way. For some operation types, additional corrective operations need to be inserted to keep the data separated or to maintain the functional correctness with respect to the unpacked execution. Finally, the results have to be unpacked at the end of the computation. Packing and unpacking are conceptually not different from what is needed for Hardware SIMD, but a different number of words needs to be combined and their respective positions inside the word are also different. The cost of this overhead should be carefully balanced against the expected gains.

Due to the extra operations, complex trade-offs come into play. Estimating the expected gains is significantly more complex than for word-width aware scheduling and assignment. Therefore, a closer look at Software SIMD is needed and this is presented in Section 9.4

9.4 Software SIMD

This section shows how significant gains in both performance and energy efficiency can be obtained during the mapping process when word-width information is exploited during parallelization. Different parallelization options are presented, comparing the sequential code to versions exploiting either SIMD supported by hardware (SIMD in the traditional sense, here referred to as Hardware SIMD) or Software SIMD. For the SIMD cases, the way data is stored initially in memory is being considered. Finally, the energy and performance estimations for each of these mappings are shown and conclusions can be drawn.

9.4.1 Hardware SIMD vs Software SIMD

Current energy efficient and high performance processors use (Hardware) SIMD (Single Instruction Multiple Data) to make use of the data parallelism available in applications, to obtain a higher energy efficiency and to boost performance. SIMD processors provide special hardware in their datapath that supports computations on a certain combination of sub-words of the same length (e.g. 1 $\times$ 64 bits, 2 $\times$ 32 bits, 4 $\times$ 16 bits or 8 $\times$ 8 bits), producing multiple separate results. The availability of multiple SIMD modes is called $multi-gauge$ in literature. This capability can be used to differentiate

SIMD processors from Vector processors that support only a single parallel execution mode (e.g. 10×32 bits).

In many cases, however, the limited number of different word-lengths that are supported requires the data widths to be rounded to the next available supported SIMD mode, which leads to a loss in efficiency. When the smallest sub-word supported by the hardware is, e.g. 8-bit, an operation on 5-bit data has to be promoted to 8-bit sub-words, which leads to a total of 24-bits that are wasted on a 64-bit datapath.

By providing more detailed word-width information to the mapping process, different and potentially more efficient combinations in mapping can be found. These detailed word-widths should not be restricted to pre-defined word-widths or rounded to the next sub-word size (of higher width) that is supported by hardware, as is traditionally done.

The *Software SIMD* technique is intended to exploit the word-width information and to deliver more efficient mappings by combining different word-widths that would not be supported by the hardware, or could even be used to make use of SIMD on a datapath that does not support this. However, this lack of hardware support requires the insertion of extra operations in specific cases, in order to guarantee the functional correctness. There is a trade-off involved here and the overhead of the extra operations turns out to be very small if the fixed point refinement is properly done. In general, a more systematic trade-off exploration is needed.

As with regular Hardware SIMD, the overhead of packing and unpacking data can be limited if the data are used together for a large number of times or can be eliminated if the data are stored packed in memory. Modern applications, especially in the wireless context, are well suited to use this technique.

Software SIMD improves performance and energy efficiency when sufficient word-length variation is present Software SIMD can directly affect the performance and energy consumption of many different parts of the platform. As with Hardware SIMD, the increase in parallelism leads to a reduction in the number of operations that are needed to execute a certain kernel for a fixed set of data. This leads to a reduced number of accesses to the instruction memory organization. If different shorter data words are stored together in a single word, the number of accesses to the data memory hierarchy (and the memory footprint) can be reduced. As less operations have to be scheduled, this leads to an increase in performance. In contrast to Hardware SIMD, where only packing and unpacking lead to overhead, for Software SIMD, the extra corrective operations that are inserted affect the gains and should be monitored carefully. When enough different word-lengths are present, that are not only a power-of-2 variation, then usually the energy gains (strongly) exceed this cost.

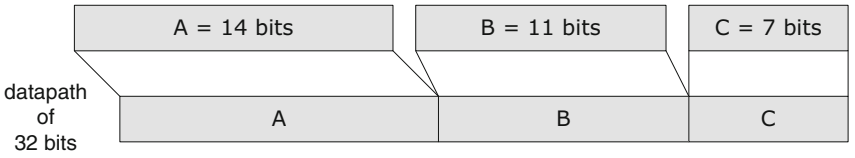


Figure 9.14: Packing heterogeneous data in order to be able to operate on them together later, using Software SIMD

Software SIMD is more flexible In addition to packing different sub-words of equal and pre-defined lengths, as is supported by the hardware, Software SIMD can exploit (combinations of) different widths that are present in applications, as was shown in Section 9.1.2. In order to operate on a combination of data elements in a SIMD fashion, the elements of arrays A, B and C, which are for example respectively 14, 11 and 7 bits can be packed into a 32-bit word (for a 32-bit datapath) as is shown in Figure 9.14. The extra bits that potentially have to be added to stop overflow from one sub-word to the next are discussed below. The additional freedom and optimization potential that are created as such can lead to the usage of SIMD in applications that cannot use Hardware SIMD (e.g. because the minimal word-width that is required is larger than the supported sub-words). Even in mappings that already make use of traditional SIMD, a more efficient mapping could be obtained when making use of the more flexible Software SIMD. This extra flexibility leads to many more potential SIMD combinations, but increases the complexity of deciding which parallelization to use and to estimate the expected gains of applying the Software SIMD.

9.4.2 Enabling SIMD without hardware separation

Operating on different sub-words in parallel, without specific hardware support to keep the data of these sub-words separated, can not be done without the insertion of extra operations. What exactly is required depends on the type of operation and on the characteristics (in this case with respect to the word-width) of the data. The combination of packing, execution on a regular (non-SIMD) datapath and the insertion of the corrective operations where needed, is called $Software-SIMD$.

This section discusses the enabling techniques and restrictions. In the following discussion, we assume that Software SIMD will be applied to selected kernels of an application. Therefore, the scope is the algorithm that makes up the computation of a certain kernel.

Saturation behavior without hardware support As has been discussed in Section 9.1.1, some applications require the implementation of saturation

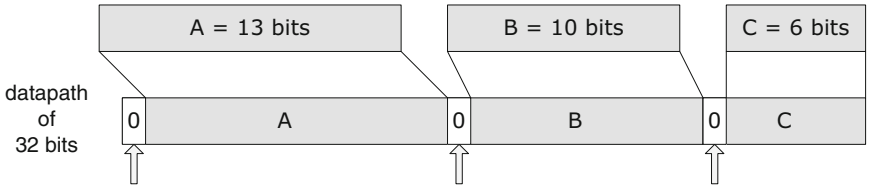


Figure 9.15: Extra guard bits (set to 0) are added to be able to handle over- and underflow

behavior in order to prevent unwanted overflow and underflow. As with hardware, the Software SIMD approach can support saturation at an extra cost. In order to make this test possible and correctly handle over- and underflow when required, an extra guard bit is added to the left of every sub-word, as shown in Figure 9.15.

Depending on the instruction set of the used processor, the implementation of the saturation behavior can be more or less costly. An example of saturation for overflow and underflow will be presented below, for addition and subtraction respectively.

Signed numbers scaled to unsigned The special treatment of the sign bit for some operations (e.g. arithmetic shift) complicates the application of Software SIMD. Therefore, we here assume that all signals are scaled to be positive (as was discussed in Section 9.1.1) and the parallelization occurs on unsigned data. Standard truncation rounding behavior rounds all numbers towards negative infinity, so there will be no additional effects on the accuracy when scaling a signal to be positive. Additional care has to be taken when rounding towards 0 is assumed (requires extra hardware support), which is not supported here. The different rounding behavior has to be taken into account, in order not to violate the application requirements.

9.4.2.1 Corrective operations to preserve data boundaries

Operating on different sub-words in parallel without using specific hardware support is only possible if the effect of the various operations of an algorithm on the separation of the data can be controlled. However, different operations that are executed on these packed words will have a different effect on the separation between the results. In order to keep the computations of the algorithm unchanged, the results of the computations should still be functionally correct and meet the application requirements.

The compute-intensive kernels of our target applications domains, namely wireless communication and multimedia, (mostly) consist of a relatively

small set of different operations: arithmetic operations like additions/ subtractions, shifts, multiplications, logic operations like AND, OR, XOR and relational operations like equal to, greater than, less than. Some of these operations will be directly supported by the processor instruction set and special modifications could be made in order to support the Software SIMD variants better (examples will be given below). However, in this chapter, it is assumed that a *standard* instruction set is used and that it is not extended specifically to support the proposed optimizations. Therefore, some operations need to be decomposed into the operations available in the processor instruction set, which changes the overall cost of e.g. using saturation. The example implementations that will be shown below assume a *standard* instruction set. If specific other operations are available to support specific target applications, these operations might be of use to reduce the cost of Software SIMD operations. For example the Altivec Instruction set [Fre] supports very flexible vector permute operations to do sub-word manipulations.

Addition Performing an addition on the packed data is rather straightforward. A schematic example is shown in Figure 9.16 for a set of data that is guaranteed not to cause overflow. This means that the word-width that is assigned to the different sub-words (A, B, C) is larger than the instantaneous values (represented by the gray boxes). The fixed point refinement guarantees, as in the case of the normal SIMD, that no overflow beyond the range supported by A, B and C will occur during the execution of the algorithm and no extra guard bits are required in this case. Potential overflow bits (represented as a black box in the result of Figure 9.16) will still stay inside the bit-range that is reserved for each data item.

If the word-width that is required to make this guarantee is too costly, a smaller width can be used and saturation can be implemented to catch the

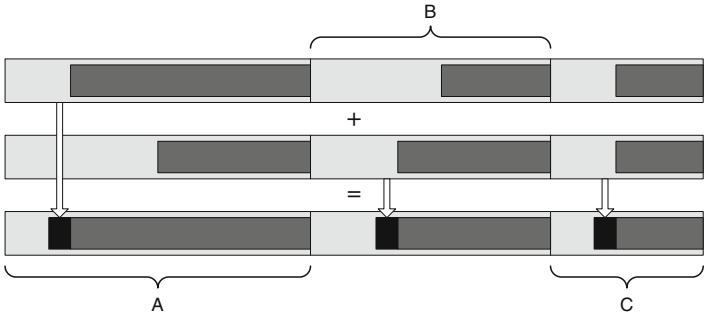


Figure 9.16: Schematic representation of the Software SIMD addition, in which the instantaneous values are smaller than the representable range of the respective sub-words. Therefore, the extra bits that are potentially generated by the addition are still within this range

occasional overflow (at an additional cost). The following example shows how this can be implemented using one extra guard bit and some extra operations. This leads to a trade-off with spending more bits on the sub-word (less chances for overflow to occur) vs. the cost of the overflow detection and how often the more expensive corrective action has to be executed. Alternatively, the scenario approach [Pal07] can be used to limit the input word-width from the start and e.g. to meet the SNR/BER requirements of a wireless communication algorithm without unneeded overhead, as is shown in [Nov08, Nov09].

Addition with overflow saturation Figure 9.17 shows three packed words: inputs ①, ② and result ③ , which consist of data a, b and c, d and e, f respectively and guard bits q, r, s for the first sub-words and x, y, z for the second sub-words. The guard bits are 0 by default.

When performing an overflow-sensitive operation (and only then), saturation can be implemented, as described below. A normal unsigned addition (for the full datapath width) is performed and followed by a test (using AND (&) and an equality comparison ==),[2] which detects an overflow in any of the sub-words. Guard bits are represented in bold, as e.g. **1**.

$$\text{standard unsigned addition: } ③ = ① + ②$$
$$\text{overflow test: } (③ \,\&\, \boxed{\mathbf{1}0\ldots0\mathbf{1}0\ldots0}\,) == \boxed{\mathbf{0}0\ldots0\mathbf{0}0\ldots0}$$

The overflow mask ($\boxed{\mathbf{1}0\ldots0\mathbf{1}0\ldots0}$) selects only the guard bits (shown in bold), which are then compared to 0. In this way, the overflow test can successfully detect the occurrence of an overflow and a corrective action can be taken if required. Different corrective actions can be generated at compile

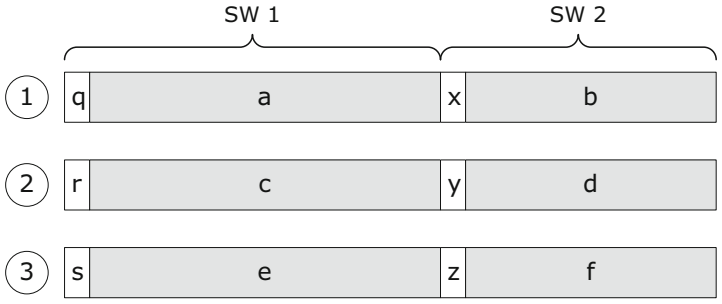


Figure 9.17: Schematic representation of the packing of data into sub-words SW1 and SW2, separated by extra guard bits

[2]Note that bitwise operators are represented by a single symbol, e.g. &, while the logical conjunction AND would be represented by the double symbol &&.

time and are stored in the loop buffer. At run-time, the required saturations can be performed, using the loop buffer's local controller (as described in [Jay02a]). If no local controller is available, a branch to a sub-routine that handles the saturation or the use of predication to nullify the saturation of sub-words that are not needed can be used, but this will be less efficient. The test itself requires a mask operation and a (regular full-word) comparison only.

If the test fails, an overflow is detected and all guard bits are separately compared to 0 in order to check which sub-word caused the overflow. Many DSPs support instructions that would be able to handle this efficiently (e.g. get position of most significant non-zero bit). A special instruction could be added to handle the saturation very efficiently, by directly saturating the sub-words that have non-zero guard bits. If no such special operations are supported, separate mask operations can select all guard bits one by one. The $guard{-}bit\ select\ masks$ contain a 1 for a single guard bit, while all other bits are 0 (e.g. $\boxed{\mathbf{1}0\ldots0\mathbf{0}0\ldots0}$ for the first guard bit). If a specific guard bit is found to be 1, an overflow is detected for that sub-word and the corresponding sub-word is saturated (using an OR (|)). The other sub-words are left untouched. As multiple sub-words could generate an overflow, all guard bits have to be checked. However, these comparisons can be done in parallel.

$$\text{overflow test for SW1: } (③ \,\&\, \mathbf{1}0\ldots0\mathbf{0}0\ldots0) == \mathbf{0}0\ldots0\mathbf{0}0\ldots0$$
$$\text{saturation of SW1: } ③ \mid \mathbf{1}1\ldots1\mathbf{0}0\ldots0$$

After all sub-words that generated an overflow have been saturated, the overflow bits are set back to 0.

$$\text{reset guard bits: } ③ = (③ \,\&\, \mathbf{0}1\ldots1\mathbf{0}1\ldots1)$$

It should be noted that the occurrence of an overflow should $only$ be checked after overflow sensitive operations (this information is propagated down the fixed point refinement or from the algorithm design) and that the overflow test is relatively cheap. It should be stressed here that this will not be required e.g. for every addition in the algorithm. Only when an overflow effectively occurs, the saturation penalty has to be paid.

Subtraction For a subtraction, the assumption that data is scaled such that the algorithm operates on (and produces) unsigned, positive data, results in subtractions where the relative size of both operands is known and the result will always be positive. This assumption can, next to the fixed point refinement, be partly enabled by the use of correlation information on the data at compile time and limited explicit tests at run-time (scenario detection). Using the example of Figure 9.17, this means $a \geq c$ and $b \geq d$. In this

case, the assumption that all data will be positive is always respected and the subtraction can be performed as with a normal subtraction of datapath width.

standard unsigned subtraction: ③ = ① - ②

However, if this is not always guaranteed, there is a risk that the result of the subtraction for any sub-word would have a negative result, which would destroy the separation of data. This is because the standard subtraction will "borrow" a bit from a sub-word on the MSB side and produce a wrong result. To stop this from happening, two approaches can be taken. Firstly, underflow saturation can be implemented, meaning that a negative result will be rounded to 0. Secondly, a test can be performed to check the relative size of sub-words, before performing the subtraction and required corrective actions can be taken. Example implementations of underflow saturation and relational operations are shown next.

Subtraction with underflow saturation In order to correctly detect underflow, the guard bits of the first operand are set to 1. This again should only be done in those parts of the algorithm that have a risk to underflow.

set guard bits to 1: ① = (① | $\boxed{\mathbf{1}0\ldots0\mathbf{1}0\ldots0}$)
perform standard subtraction: ③ = ① - ②)
underflow test: (③ & $\boxed{\mathbf{1}0\ldots0\mathbf{1}0\ldots0}$) == $\boxed{\mathbf{1}0\ldots0\mathbf{1}0\ldots0}$

For the same packing as shown in Figure 9.17, the following cases are possible:

if $a > c$ and $b > d$: $e = a - c$, $f = b - d$ and $s == 1$, $z == 1$ (no underflow)
if $a < c$ or $b < d$: guard bits x and q stop the $borrow\ from\ MSB-side\ sub-word$, but the result is wrong

If the underflow test fails (overflow detected), the same approach is taken as with the overflow for additions: test which sub-word caused the overflow and saturate it, set it to 0. Finally, the guard bits are set back to 0, their default value.

reset guard bits: ③ = (③ & $\boxed{\mathbf{0}1\ldots1\mathbf{0}1\ldots1}$)

Logical operations and masking Logical operations, like NOT, AND, OR, XOR, etc., as have been used above, are bitwise operations. Therefore, there

is no need for special hardware support and these operations are supported by the Software SIMD approach as is.

The masks that are used together with the logical operations in the example implementation of saturation above, can be generated at compile time and stored in memory or can be passed as part of the operation (as an immediate). A limited number of different masks are required.

Relational operations Relational operations like $equal\ to$, $greater\ than$, $less\ than$ etc. require specific hardware to separate the comparison of the different sub-words and can not directly be supported when using Software SIMD. However, they can be decomposed into supported operations.

Different kinds of relational operations can be supported. One type produces a mask, in which bits at specific places indicate the outcome of the comparison. Figure 9.18 shows an example implementation for relational operations that produce such a mask, based on the subtraction. Other relational operations, that e.g. combine the largest sub-words of both inputs (compared one to one) into the output, can be implemented using such masks. The top of the figure defines the input words, with their respective sub-words. Additional information, to be used in this example, is presented on how the sizes of corresponding sub-words relate to each other. The rest of the figure indicates how $\geq$, $\leq$ and $==$ can be implemented, producing the result of the respective comparison for each sub-word in the guard bits. Finally, $<$ and $>$ can be generated as a combination of the above, using a logical XOR operation.

Even though the relational operations can be implemented as presented above, it should be noted here that the usage of this kind of operation in a Software SIMD kernel should still be avoided, as the operations following the relational operation will probably depend on the result of the comparison. However, a Hardware SIMD implementation suffers from the same problem. Therefore, speculative execution of both paths or the use of algebraic transformations to reduce the amount of relational operations should be considered when enough design time is available. As a (more costly) fall-back solution, the operations that depend on the comparison can always be performed on the unpacked data.

In an ASIP implementation, the cost of relational operations can be heavily reduced by adding specific operations to the instruction set that enable the cheap generation of masks. For example providing the position of the guard bits ($\boxed{10\ldots010\ldots0}$) and the result of a comparison as shown in Figure 9.18 as inputs, a mask can be generated with 1 bits for all sub-words that have a guard bit equal to 1 and 0 bits for the other sub-words. Using these masks, e.g. the largest sub-words of two input vectors can be combined into an output vector efficiently. A more thorough evaluation of potential instruction set extensions is part of future work.

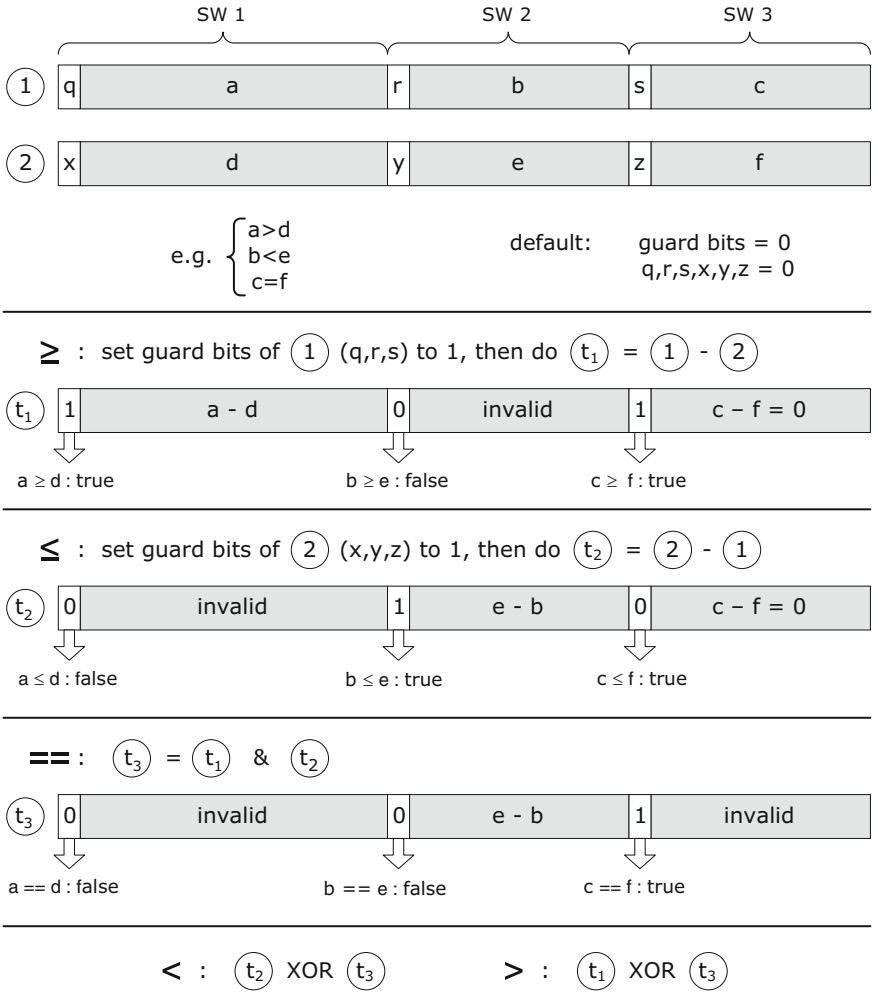


Figure 9.18: Example implementation of Software SIMD relational operations, based on subtraction with underflow detection

Left shift As all data is assumed to be positive, only logical shifts have to be considered. Arithmetic shifts preserve the sign bit for signed numbers, which is not required here. However, a difference still has to be made between three types of logical left shifts, with respect to the way they are used.

For the first type (Figure 9.19), a shift to the left over x positions, the instantaneous input data of the respective sub-words is not shifted out of the representable range of each sub-word. This type is supported for use in a software-SIMDized kernel as is and does not require any special extra operations.

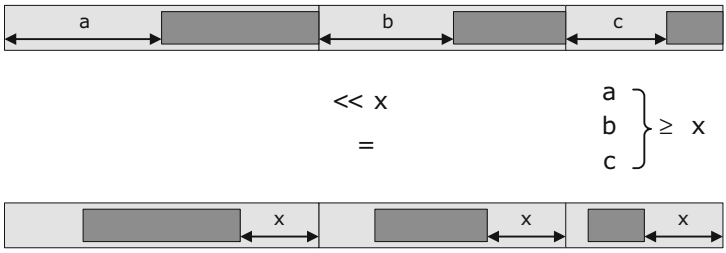


Figure 9.19: Software SIMD left shift with data within MSB range

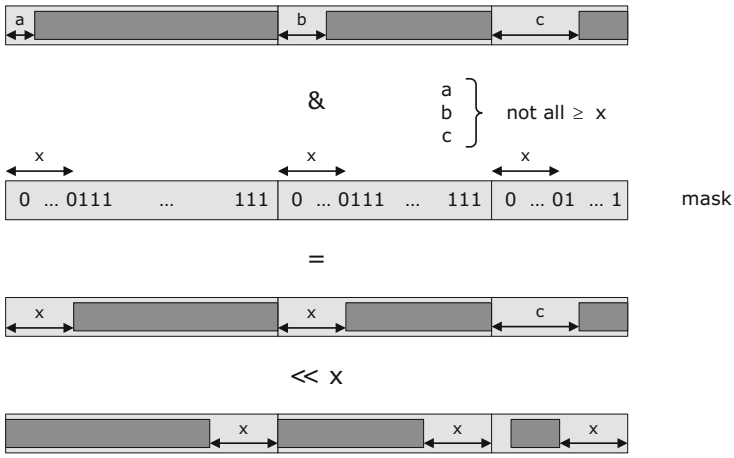


Figure 9.20: Software SIMD left shift with data out of MSB range

This excludes a special second type of left shift that requires truncation, as is shown in Figure 9.20. In some cases, the shift operation is used to select a certain bit range by shifting the data partially out of the word, thereby truncating the original data on the MSB side. This type of shift can be seen as a bad programmer's practice and should be avoided, but it can still be supported. As there is no hardware to enforce the boundaries between the different sub-words, an extra mask operation is required to prevent bits from one sub-word to be shifted into the next. This can be achieved by adding a logical AND with a mask, in which the bits that correspond to the to-be-truncated range are set to 0 and all other bits are set to 1. After this extra operation, the shift can be performed using the normal shift operation.

Alternatively, the mask operation to force the to-be-truncated range to 0 can be moved after the shift. This enables combining multiple of these operations into a single mask, if some later shifts or other operations also need mask operations. This kind of transformations helps to reduce the overhead.

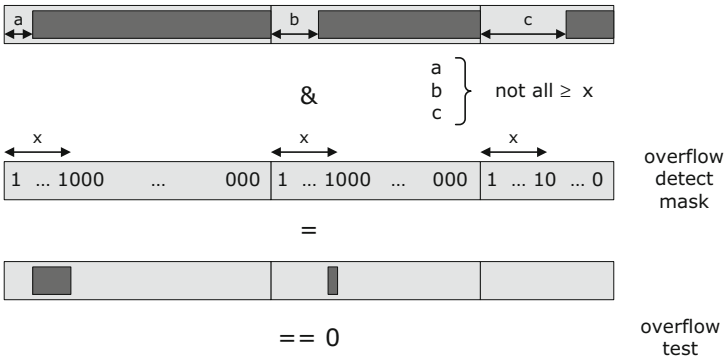


Figure 9.21: Software SIMD left shift saturation detection

As with the addition, a left shift can potentially lead to an overflow. In that case, it can be combined with saturation behavior, which makes it the third type of shift. An extra test is needed to detect if an overflow has occurred and for which words the saturation should be applied. The overflow detect mask selects the x leftmost bits of each sub-word, when shifting left over x positions, as shown in Figure 9.21. The result of this mask is compared to zero to detect an overflow in any of the sub-words (the two leftmost sub-words will require saturation in the example of Figure 9.21). If the test succeeds, meaning no saturation is required, a normal left shift is performed. If the test fails, extra comparisons are done to detect which words need saturation. Then, as for the shift with truncation, a mask is used to remove the x leftmost bits of each sub-word, followed by the shift and the saturation of the offending sub-words (as with the addition).

Again, the overflow detection is cheap, requiring only one AND and one comparison. If no overflow is detected, a normal shift can be performed. If overflow is detected, a more expensive correction is required.

Right shift Finally, a logical shift to the right can be implemented in a similar way. In this case, the extra mask operation is always required to prevent bits from the LSB side of one sub-word to enter the MSB side of another one. As is shown in Figure 9.22, the logical AND operation is inserted to force the to-be-shifted right bits to zero, before (or after, as explained above) the actual shift operation is being done.

If many shift operations are required, in ASIPs, the addition of a dedicated shift slot would be preferred to reduce the cost of the shift operations. This unit is simpler than the combined shuffle/shifter [Rag07b] that is needed for pack/unpack operations.

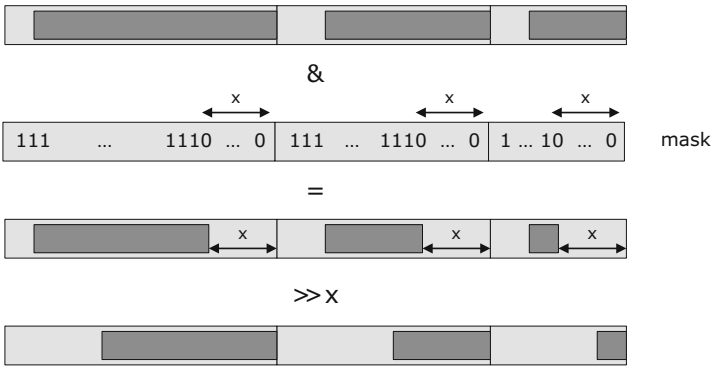


Figure 9.22: Software SIMD right shift

Note that the above described implementation of left and right shifts depends on the packing. It is possible to pack the instantaneous data into the MSB side of their respective sub-words, which results in simpler right shifts (if the shift distance does not shift the data out of the reserved width), but more complex left shifts. Depending on the context, i.e. the number of shifts of either direction that are used in the algorithm, a preference can be given to the packing style.

Multiplications are a problem Additions/subtractions, shifts and logical/ relational operations all behave in a controllable way with respect to how the separation of packed data in the input is propagated to the output. An efficient usage of those operations in a Software SIMD implementation is still feasible as long as the fixed point refinement was properly done. As was shown in Figures 9.16 and 9.19, the separation of the sub-words is maintained in the result (the same bits belong to the same sub-word).

For multiplications, however, the placement of the result corresponding to the product of the input sub-words is more complex. As can be seen in Figure 9.23, the width of the complete result of a multiplication is the double of the input width (e.g. 2×32 bits gives a 64-bit result). Typically, this result is then cast back to the single word-width afterward, to avoid increasing word-widths. However, when using a normal (full-word) multiplication for packed data, the placement of the results of the product terms is such that they need to be placed at the extreme MSB and LSB side of the packed word, in order to be able to recover the products of individual sub-words. As is shown in Figure 9.23a, a large separation in between has to be reserved to hold the cross-products that are unwanted here. These cross-products will quickly overlap with the useful products, thus making the useful data unrecoverable. For the simple example shown here, packing only 2 sub-words and for widths of a, b and c, d of x and y bits respectively, the width of the guard

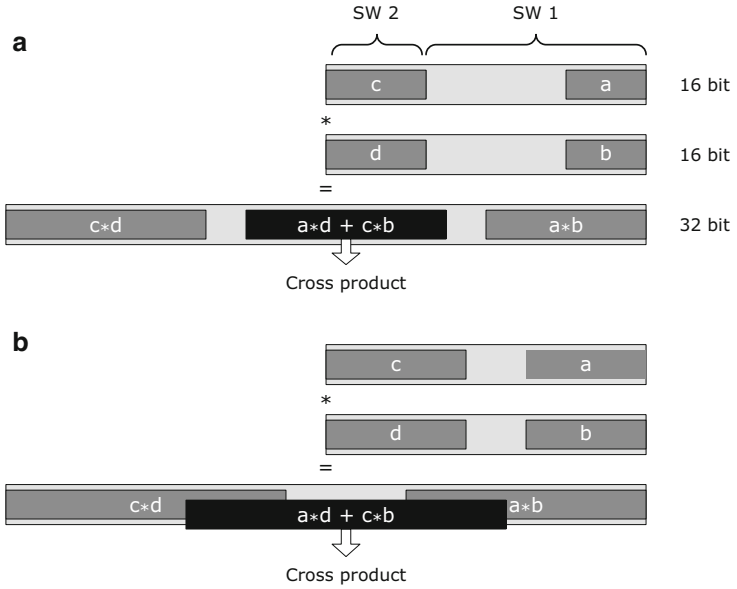


Figure 9.23: Overlap problem when using MUL and Software SIMD

band z has to respect: $z \geq (x + y + 1)$. This restricts the useful part of the datapath to less than half of the full width. For more than two packed subwords, this effect is even worse as more unwanted cross-products are placed all over the word.

This effect results in a very sparse usage of the datapath, whenever multiplications are involved, which is the absolute opposite of what Software SIMD is supposed to achieve. Therefore, related work [Eve01, Ber06] does not allow any multiplications or requires extremely large guard bands [Kra07]. As this restriction heavily reduces the applicability of this technique in our target application domain, we propose a conversion of multiplications into a sequence of shift and add operations. This conversion is discussed in detail in Chapter 10.

Also for other types of operations that are not supported by the Software SIMD technique, the decomposition approach can be followed. As very few processors support division, square root, modulo, trigonometric operations, etc. natively in their instruction set, these operations are converted into other supported operations. That can happen for example very effectively based on the CORDIC transformation [Vol59]. That is illustrated for the bioimaging application in Chapter 11. In most cases more than one conversion is possible. Therefore, it can make sense to revisit the conversion strategy in the context of Software SIMD. Apart from the multiplications, for which this aspect is discussed in Chapter 10, this is outside of the scope of this book.

9.4.2.2 Software SIMD on a Hardware SIMD capable datapath

Software SIMD can be used as a technique to exploit SIMD on non-SIMD capable datapaths. However, on SIMD datapaths, it can still be used to exploit heterogeneous sub-words or sub-words of other sizes than the supported powers of 2. Whenever the required word-widths match the word-widths supported by the hardware, it will always be better to use the HW SIMD, as no extra operations are required. However, when heterogeneous word-widths can be combined, or when the required word-width differs significantly with what is supported by the hardware, Software SIMD can be used. An example of a homogeneous parallelization using Software SIMD on a Hardware SIMD capable datapath is presented in Section 9.4.3 for a simple case and in Section 9.4.4 for a more challenging case.

9.4.3 Case study 1: Homogeneous Software SIMD exploration for a Hardware SIMD capable RISC

This section starts with a theoretic estimation of the potential of Software SIMD. Table 9.1 presents an optimistic estimate (no extra overhead) of the gains that can be expected when using homogeneous Software SIMD for various potential word-widths. A datapath of 64 bits is assumed with 8×8, 4×16, 2×32 and 1×64-bit SIMD modes. The table presents the

WW	# in 64-bit DP	HW SIMD	Conclusion	If no 1×64 mode
4	12 ($12 \times 5 = 60$)	8	**50%**	Yes
5	10 ($10 \times 6 = 60$)	8	**25%**	Yes
6	9 ($9 \times 7 = 63$)	8	**12.5%**	No
7	8 ($8 \times 8 = 64$)	8	No gain	Yes
8	7 ($7 \times 9 = 63$)	8	12% loss	No
9	6 ($6 \times 10 = 60$)	4	**50%**	Yes
10	5 ($5 \times 11 = 55$)	4	**25%**	No
11	5 ($5 \times 12 = 60$)	4	**25%**	No
12	4 ($4 \times 13 = 52$)	4	No gain	Yes
13–15	4	4	No gain	Yes
16	3 ($3 \times 17 = 51$)	4	25% loss	No

Table 9.1: Comparison of the theoretic maximal performance gain (without additional corrective operations) for data with a certain word-width (WW) for Software SIMD (# in 64-bit DP) and Hardware SIMD (HW SIMD), assuming a 64-bit SIMD datapath with 8×8, 4×16, 2×32 and 1×64-bit modes. The expected gain (or loss) is given, next to the availability of that option if no 1×64 mode would be available

theoretically achievable gains for the best case: firstly, when no additional corrective operations are required and secondly, if there are no dependencies that restrict the parallelization.

Table 9.1 presents the evolution in the expected gain for varying data widths. The first column, WW, represents the required word-width, ranging from 4 to 16 bits. The second column indicates which Software SIMD mode will be selected, when operating on a 64-bit datapath and how many words can be operated on in parallel. The HW SIMD column indicates to which Hardware SIMD mode this can be compared. This leads to a projected gain, presented in the next column (Conclusion). An additional column indicates the availability of that specific solution on a datapath that only supports 8×8, 4×16 and 2×32-bit SIMD modes. In that case, the sub-words have to be aligned correctly with the 2×32-bit SIMD mode.

The table shows that even if homogeneous Software SIMD is applied on a datapath that supports SIMD in hardware, for minimal word-widths of 4, 5, 6, 9, 10 and 11, significant performance gains of 11–50% could be achieved by using Software SIMD. For all other cases, the usage of Software SIMD would only make sense if the hardware would not support SIMD. The omission of the full 64-bit mode rules out the solutions for 6, 10 and 11 bits.

The parallelization options presented in Table 9.1 assume that a guard bit is added to every sub-word. If no relational operations or saturation are required, the guard bits can be removed, which leads to improvements with respect to the table as potentially more sub-words can be packed together. For example for a word-size of 9 bits, 7 sub-words can be packed, compared to only 4 in the Hardware SIMD case, which leads to a maximal improvement of 75%. Without guard bits, there will be no cases that have a loss with respect to the HW SIMD.

Software SIMD for the MIMO benchmark Following the predicted performance improvement of 50% when performing Software SIMD for a word-width of 9 bits, in the following part of this section, we will present an energy estimation for the platform and benchmark application that has been described in Section 9.2.3, namely a compute intensive part of the MIMO benchmark on a 64-bit Hardware SIMD-capable in-order RISC. For this realistic wireless communication kernel, the energy consumption (using the word-width aware energy models introduced in Section 9.2) and performance (including potential packing effects) are estimated for five different Hardware and Software SIMD code versions. The selected benchmark in this case only consists of additions, subtractions and multiplications. After the multiplications have been converted (see Chapter 10), only additions, subtractions and shifts are required. The original implementation does not require saturation and is using 16-bit data. We assume that a re-quantization of the data (taking into

account the specific requirements of the target system) requires only 9 bits per word. For this specific example, no extra Software SIMD specific corrective operations need to be inserted and the potential of this technique can be estimated. This means the impact on performance is as predicted in Table 9.1. The experiment of this case study is the result of a manual conversion of the original non-SIMD code (compiled) to the various SIMD versions.

The potential energy gains of exploiting word-width information are demonstrated by presenting five different mappings:

1. No SIMD: In this case, no SIMD is used on the processor (as a pessimistic baseline case), and all data are aligned in the data memory to 64-bit boundaries (only 9 bits out of 64 contain the data).

2. Hardware SIMD: The second reference case assumes that the Hardware SIMD provided by the datapath hardware is exploited and all data are aligned in memory to 16-bit boundaries (so in each sub-word, that actually contains only 9-bit data, the 7 MSB bits are 0). In this case, the data are pre-packed when they are read from the memory in an optimal way for the datapath ($4 \times$ 16-bit mode) and also can be stored back packed.

3. Hardware SIMD with Pack/Unpack: If the data is not pre-packed in memory, the Hardware SIMD capability of the datapath can still be exploited, but extra pack and unpack operations have to be performed in order to transform the data layout. In this case, the datapath operates in the $4 \times$ 16-bit mode, but the data layout is assumed to be aligned to 64-bit boundaries (only 9 bits out of 64 contain the actual data). To transform the data layout, we combine four loaded words into one word, using a pack unit. At the end of the computations the words are unpacked and stored back using the original data layout.

4. Software SIMD: A more efficient mapping can be obtained by directly operating on the 9-bit sub-words. Assuming that data are pre-packed in memory according to 10-bit boundaries (9-bit data + 1 guard bit), 6 sub-words of 9 bits can be operated on in parallel, filling the 64-bit datapath almost completely. The guard bit is not needed for the current kernel, but is included to support relational operations or saturation elsewhere in the application and to avoid re-packing later. Without the guard bit, one more sub-word can be fitted in for this case (for the same overhead), which leads to larger gains. The pre-packing assumption is realistic if the producer code can be modified also.

5. Software SIMD with Pack/Unpack: If the fixed point refined data are not pre-packed, or if they are only used in the specific packing that is needed in this kernel once, the packing/unpacking overhead cannot be neglected. In this case, the extra (shuffle/shift) operations, needed

to transform an initial data layout of 64 bits to the packed version of the previous case, are included. This introduces five pack operations and six unpack operations for every Software SIMD enabled operation. Most algorithms will provide the opportunity to re-use the packed data inside the kernel itself, so this packing overhead is rather worst-case.

Figures 9.24 and 9.25 show the energy breakdown and performance of all mapping variations described above, using word-width aware models in all

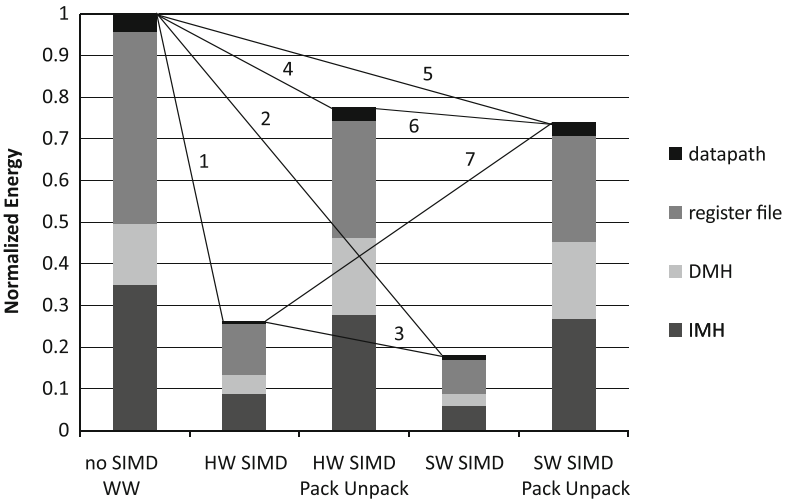


Figure 9.24: Energy breakdown of different mapping variants of the MIMO kernel, using different SIMD options and data layouts

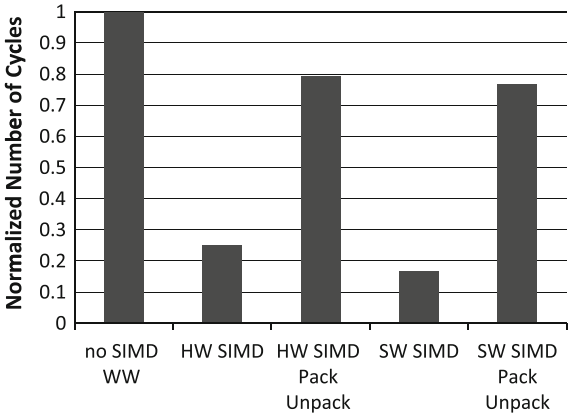


Figure 9.25: Performance results of different mapping variants of the MIMO kernel, using different SIMD options and data layouts

cases (for the energy plot). The results have been normalized to the energy consumption and performance of case 1, which serves as a pessimistic baseline case. For case 1, no SIMD is used on a 64-bit wide datapath, which leads to a high energy cost and a bad performance (as is to be expected). In this case, every data item that is needed during the computation is loaded from the memory to the register file, operated on and eventually stored back individually.

Using the Hardware SIMD capabilities provided by the hardware, and assuming a SIMD optimized data layout, the energy consumption is reduced by more than 70% (1 in Figure 9.24) and performance is improved with 75% (for the same 2 bars in Figure 9.25). This is the result that can be achieved by a state-of-the-art approach, using a good mapping. In this case, the number of loads from the memory to the register file is heavily reduced by loading multiple data in parallel (four data items of 9 bits, each promoted to 16 bits in the full 64-bit datapath width), which leads to energy savings in both the register file and the memory. The number of computational operations is reduced by the same factor, which leads to less accesses to the instruction memory.

By exploiting extra word-width information and performing Software SIMD, the energy consumption can be reduced even more. When, as in the previous case, the data layout has been optimized ($SW-SIMD$), a reduction of over 80% compared to the baseline (2) or over 30% compared to $HW-SIMD$ (3) can be achieved. Note that if a state-of the-art SIMD implementation would serve as a baseline, this improvement of over 30% is very significant. Performance is improved with 83% compared to $no\ SIMD\ WW$ and with over 33% to $HW-SIMD$. In this case, more data items are packed together in one word (6 data items of 9 bits, each with a 1-bit guard band, filling the 64-bit datapath width), which lead to even less accesses to the data memory and the register file and to less operations and accesses to the instruction memory hierarchy.

When the data-layout has not been optimized, or when different packings are needed in different parts of the application, the overhead of the pack and unpack operations cannot be removed completely or neglected. When transforming the data layout in order to use Hardware SIMD ($HW-SIMD$ $Pack-Unpack$), the number of datapath operations that is performed on the packed data is still smaller than for $no\ SIMD\ WW$, but a high number of pack and unpack operations are added to make this possible, which leads to a larger energy cost in the datapath itself. The number of accesses to the register file are, however, overall still reduced. The number of accesses to the data memory and the energy cost of this part are exactly the same, as the original and final data layouts are the same. Compared to $no\ SIMD\ WW$ the energy consumption is still reduced with over 20% (4) and performance is improved with about the same amount.

By adding extra word-width information and performing Software SIMD under the same constraints as for $HW-SIMD\ Pack-Unpack$, an additional 3.5% (5) can still be gained, on both energy and performance. In this case, the number of accesses to the register file and the number of operations is still marginally reduced, but even more pack and unpack operations had to be added, which leads to an overall gain of just a couple percent (4.6% better than $HW-SIMD\ Pack-Unpack$, (6)). However, this gain is still relevant, as it corresponds to an improvement for the full platform, including the processor and the data, instruction memories (DL1 + IL1).

In case the data layout has been optimized for the Hardware SIMD and if this can not be changed, one could still decide to re-pack the data in order to benefit from the extra parallelism of the Software SIMD approach. However, as the overhead of the pack/unpack operations is large, this would lead to a penalty, both in energy and performance (7). As embedded systems mostly allow the designer to modify the data producer, the packing can be changed there to prevent this from occurring.

It should be noted here that without going through the effort of modeling the effective word-width to estimate the energy, gains would look different here. Without word-width aware energy models, the energy cost of the datapath would be overestimated for $no\ SIMD\ WW$, as has been shown in Figure 9.6, and less in the other mappings, as there the datapath is much better filled. This would lead to larger expected gains, which in reality would not be present, when comparing the sequential with the parallel versions. In some cases, this would justify more expensive data layout transformations, which in the end might lead to an overall energy loss. As is described in the analysis of the appendices of [Lam09], the variation of the energy consumption with varying word-widths for non-datapath components is currently rather small, but can be improved by using more optimized design methods. This would lead to larger variations and would reduce the correlation between Figures 9.24 and 9.25. Currently, the most visible part of the energy variation is related to the difference in accesses to the register file, the DMH and the IMH. These differences are the direct result of operating on a different number of elements in parallel, which is reflected directly in the performance. Therefore, the overall $shape$ of both figures is the same.

9.4.4 Case study 2: Software SIMD exploration, including corrective operations, for a VLIW processor

The second case study, presented in this section, presents a more complex FFT (Fast Fourier Transform) kernel, taken from the TI DSP library [TI09d]. The benchmark performs a mixed radix forward FFT. It is used in the radix 4 mode for a set of 1,024 input samples, which are assumed to

represent one OFDM (Orthogonal Frequency Division Multiplexing, a wireless communication standard) symbol of a wireless communication. The multiplications have been converted (see Chapter 10) for some code versions (MUL-converted) in order to enable the Software SIMD.

Experimental setup In contrast to the previous case study, this kernel has been Software SIMDized and optimized for an embedded eight-slot VLIW processor (clustered heterogeneous VLIW, with two clusters of four slots each, as shown in Figure 3.6). In this case, the addition of various corrective operations was required in order to prevent unwanted overflows. In selected parts of the algorithm, overflow tests and saturation behavior for additions was implemented as has been described in Section 9.4.2.1. These extra operations and the fact that the target processor in this case is a parallel processor, result in significant differences with respect to case study 1. Given the fact that differences in the energy consumption of today's processors are dominated by the change in accesses to the register file, DMH and IMH (as was concluded in the previous case study), this section will only present performance estimations. When architecture modifications, as proposed in Chapter 3, are implemented, the energy breakdown will be more balanced and a separate estimation of the impact on the energy consumption will be needed. This is part of future work.

Parallelization exploration Different code versions have been constructed in order to compare various Hardware SIMD and Software SIMD implementations. As in case study 1, the pack/unpack operations have been separated from the kernel in order to be able to estimate the expected gains when a modification of the data-layout is possible. Because the conversion of the multiplications can have a beneficial effect on the Hardware SIMD version or on the sequential version (due to the parallel slots of the VLIW in this case), the conversion has been applied independently of the Software SIMD technique as well. The presented results have been generated using the COFFEE tool flow [Rag08b].

Figure 9.26 presents the results of this experiment. The original benchmark, performing an FFT on 1 OFDM symbol at a time (no SIMD), is taken as the baseline. The other results are normalized to this case and are presented as cycles per symbol. The sequential MUL Converted code shows a slight increase in performance, as the strength reduction produces operations that can be scheduled on more slots, which lead to a performance improvement. In this case, an exact conversion was selected, which introduces multiple operations per removed multiplication. As a result, the performance improvement is slightly less than 3% in this case (for the details on this effect, see Chapter 10).

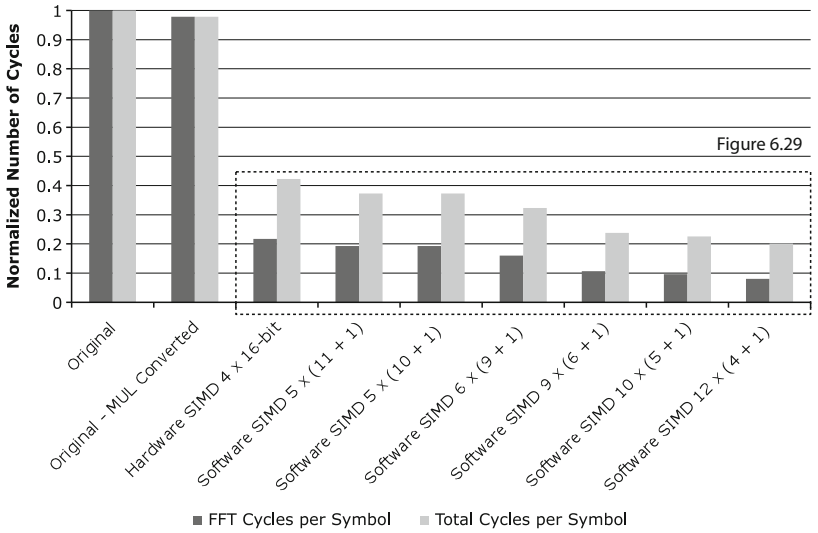


Figure 9.26: Comparison of different FFT implementations, varying the number of symbols that are computed in parallel. The results have been normalized to the performance of the original, sequential code

The original implementation operates on 16-bit data and Hardware SIMD version has been parallelized across four symbols in order to fill the 64-bit datapath. As expected, a four-way parallelization improves the performance significantly. In this case, it leads to a gain of 78% when compared to the baseline or of 57% when including the overhead of the pack/unpack operations.

Software SIMD versions for data widths out of the interesting range $\{4, 5, 6, 9, 10, 11\}$ have been constructed, as predicted by Table 9.1. Figure 9.27 presents the rest of the results, normalized with respect to the Hardware SIMD version (including packing overhead). The different Software SIMD versions operate on an increasing number of symbols in parallel, ranging from 5 to 12, for word-widths of 11–4 bits in parallel. In order to support the Software SIMD, an additional guard bit has been added to all sub-words. The guard bit is used to detect overflows, implement saturation and some types of special shift-add instructions, as discussed in Chapter 10. Even though more pack/unpack operations are added moving to more parallel Software SIMD versions (there is a slight increase in the ratio between the FFT cycles and the total cycles), there is still a gain when moving to more symbols in parallel.

As no properly quantized data was available for real OFDM symbols, randomly generated input data has been used. However, the reported performance results are obtained from valid schedules and are reported as the required number of cycles per symbol for comparison. In order to cope with

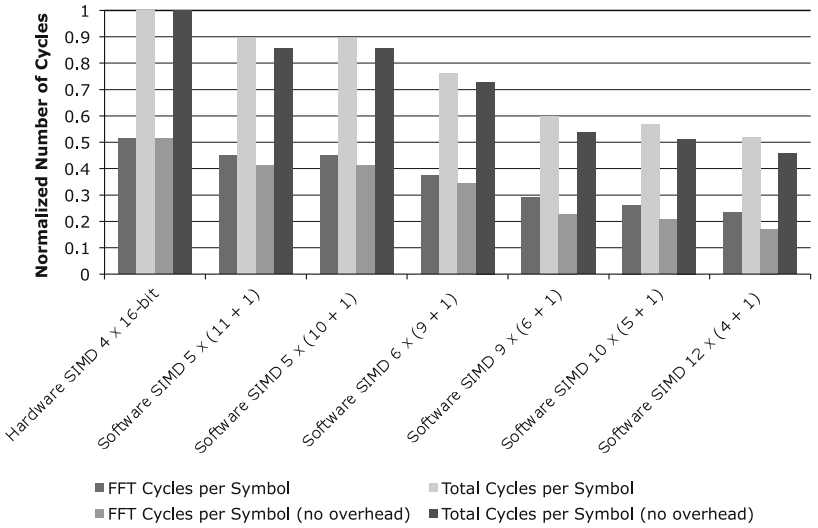


Figure 9.27: Comparison of different parallel FFT implementations, varying the word-width of the input data and thereby the number of symbols that are computed in parallel, using Software SIMD. The results have been normalized to the performance of the Hardware SIMD version

potential overflows, saturation has been implemented for a set of additions that are potentially causing overflow (based on casts present in the original implementation). In order to get a performance estimate without using real data, the assumption has been made that on average 2 sub-words out of 5 or 6, 3 out of 6, 4 out of 9 or 10, and 5 out of 12 would require saturation (arbitrary choice, with constant ratio between 0.4 and 0.5). Based on these average saturation requirements, the corresponding number of required instructions has been inserted. The gains when moving to more parallel implementations are reducing (e.g. moving from four-way to 12-way, operating on three times more symbols in parallel, only reduces the FFT cycles per symbol with a factor 2.2, from 51% to 23% or a factor 1.9, from 100% to 52% when including packing overhead). The improvements are in this case smaller than the theoretically achievable best case (a factor 3 for the same example), due to the overhead of the saturation operations. This best case, without overhead, is included in Figure 9.27, as a reference.

Due to the parallel nature of the processor, the extra operations could fit into the empty slots of the schedule. If this happens, the performance degradation will be smaller. As the FFT kernel can be heavily optimized, the produced schedule is already very dense and some extra operations do cause a longer schedule, which leads to diminishing returns when moving to more parallel versions.

As exploiting smaller word-widths in some cases will lead to more repacking in between program phases (as has been explained in Section 9.1.1), the overall overhead will increase for the smaller modes and a sweet-spot will exists.

In this case study, all pack/unpack operations are performed using shifts and bitwise AND operations. In case of hardware SIMD, in some architectures special pack/unpack units can be used that support single cycle pack and unpack corresponding to the different hardware SIMD modes (with limited flexibility). In case the Hardware SIMD version of Figure 9.27 would make use of those operations, the pack/unpack overhead would reduce by a factor 2, which corresponds to a reduction of the total cycles per symbol to about 75%. In that case, exploiting word-widths of 10 and 11 bits, which corresponds to a Software SIMD parallelism of 5, would not lead to any gains. Providing the equivalent, more efficient support for pack/unpack for Software SIMD would require somewhat more complex pack/unpack units, but would improve the Software SIMD results in a similar way, but with larger relative gains, as the initial overhead would be larger to start with. A detailed study of that more complex pack/unpack is outside the scope of this book.

The SIMD versions compute the FFT for different OFDM symbols in parallel. These symbols can potentially have different word-width requirements. This can be the case when they, e.g. belong to different wireless communication channels that have different channel conditions. Therefore, more or less bits are being assigned to the symbols (as is discussed in [Nov08]). Using Software SIMD, these variable word-width requirements can be taken into account in the implementation and the FFT of symbols of different word-width can still be computed in parallel, which is not possible with a traditional SIMD approach. A number of combinations that are likely to occur (called scenarios) can be generated at compile time and can be selected at run-time. More details on the scenario approach can be found in [Nov08, Pal07].

9.5 Comparison to related work

Word-width aware energy models and word-width aware optimizations Firstly, most instruction set simulators and add-ons for energy estimation, like Trimaran [Tri99], SimpleScalar [Aus02], Wattch [Bro00b] etc. provide average energy estimates based on activation count. This implies that the energy/activation does not depend on the precise data that is being operated on. In order to evaluate the specific effect of word-width aware optimizations that affect the datapath toggling, it is crucial to take the precise toggle activity inside the component into account while estimating the energy/power consumption. Secondly, for some platforms that consume a large part of their

energy in the datapath logic, that execute on strongly varying word-widths and for which this variation is not exploited by using SIMD, the word-width aware models will give a more realistic estimate of the contribution of the datapath logic to the total energy consumption. In this way, they can prevent wrong optimization decisions and contribute to the architecture exploration or mapping.

The models presented in this book allow designers to steer these optimizations and to evaluate in which context they should or should not be applied.

Other instruction set simulators like MPARM [Ben05a], which are written in synthesizable SystemC (using a detailed modeling of all components at a low abstraction level) are capable of producing detailed results, but are extremely slow and therefore not well suited for architecture exploration. Our previous work [Rag09a] takes into account toggle activity, but no word-width information, and does this only for the register file.

In this book, word-width aware models are presented that can still be coupled to instruction set simulator based energy estimation. Therefore, they preserve the fast speed that is required for wide architecture exploration or fast interactive application optimization.

Other energy estimation methods like [Jul03] model the energy needed for a certain operation, taking into account the previous operation, but the work is very processor specific and not-retargetable at a component level. The energy model described in [Sch04] illustrates that the actual energy consumption depends on various factors like degree of parallelism, number of instructions and also the data-width. This is however tuned only toward the TI's TMS320C6416 processor and is not scalable to other processors. Because it is highly specialized to this specific architecture, it cannot be used for other processors or platforms or for architecture exploration.

The presented approach can be used to generate word-width aware energy models for any processor component that can be modeled in VHDL, using a state-of-the-art standard cell synthesis and energy estimation flow.

Exploiting variable word-widths in custom hardware Reducing the word-width to the minimally required width is done extensively in hardware synthesis for custom logic (embedded accelerators) or for ASICS. Methods like [Syn08] provide bit-width aware architecture synthesis of custom hardware accelerators. Their PICO (Program-In-Chip-Out) system automatically optimizes the hardware by exploiting information about the varying number of bits that are required to represent and process operands. Others, like [Oze08] use stochastic bit-width estimation techniques that follow a simulation-based approach to minimize the area and energy consumption of custom hardware.

The work presented in this book enables designers to benefit from exploiting this information on processor centric platforms, thus reducing the need for custom processing blocks and leading to more flexible and reusable platforms.

Exploiting variable word-width to reduce register pressure, the memory footprint or the required bandwidth to off-chip memories Word-width aware register allocation is presented in [Tal03] in order to enable loading and storing data of various widths to the register file at a sub-word level. [Bro00a] discusses techniques to reduce the cost of the memory hierarchy by assigning smaller word-widths to narrow memory and larger widths to wide memories. Additionally, packing multiple smaller word-width data together before storing them to memory can be used to reduce the required memory footprint. Alternatively, [Thu08] presents a scheme for word-width aware data compression to reduce the required band-width between the on-chip and off-chip memories.

The concepts discussed in this book focus on the datapath and do not directly optimize for the memory footprint or register file pressure. However, by applying Software SIMD, the available datapath width is more efficiently filled. The resulting packed data that are stored to the register file are therefore using the full width. Enabling accesses at sub-word level (as presented in [Tal03]) can reduce the pack/unpack overhead in case of occasional operations that are not supported or too expensive for the Software SIMD approach. With respect to the memory footprint, the usage of Software SIMD will result in efficiently packed data that, if they are stored back to memory in the packed format, will result in automatic memory footprint reductions.

Exploiting variable word-widths in Hardware SIMD Some techniques enable the usage of variable word-widths in Hardware SIMD. For example the Mpact processor [Fol96] supports various SIMD modes for homogeneous, non-power of 2 word-widths, namely operating on sub-words of 9, 18, 24 and 36 bits. However, this type of processors still supports only a limited number of combinations that should be tuned to the application domain, which in this case was DVD decoding. A more flexible approach (with respect to word-widths) can be found in [Bal98], that proposed an arbitrary precision arithmetic for a SIMD adder. However, a proposal is presented only for additions by adding an additional gate to stop the carry at every bit. No solution is presented for other operators, which heavily restricts the application of this approach in real kernels or applications. True support for variable width SIMD is proposed in [Ann90] by using a bit-serial approach and emulation in software. However, the energy efficiency and performance implications of this approach are significant (for both energy and performance) as the

underlying hardware is effectively reduced to a width of 1bit only. The extra control and storage that is needed to cycle the data through is resulting in an expensive component. The emulation approach is reported to achieve significant speed-ups with respect to the fully bit-serial approach, but is not compared to bit-parallel implementations. A related approach, using multi-precision hardware, in which a small word-with (e.g. 4 bits) is used to implement wider operations, trades off some of the flexibility of the complete bit-serial approach for more performance and energy efficiency, but it again lacks the flexibility to adapt to non-supported widths.

Exploiting variable word-widths as Software SIMD The direct related work in Software SIMD is very limited. To the best of our knowledge, the concept was first mentioned by two very short online articles that briefly discuss the benefits of Software SIMD. [Eve01] describes a very basic approach that creates very wide guard bands by operating on even and odd bytes separately. In order to generate those separate words, extra mask operations are used, and the results are finally combined again into one word. The article does not cover other operations than the addition and only targets the emulation of byte by byte SIMD on processors that do not support any type of SIMD. [Ber06] presents an approach that is very much related to the approach of this book. They present a simple example of Software SIMD for a logical shift and an addition and propose the support of signed operations, but only for additions, at the cost of extra guard bits.

Very recently, [Kra07] presented some early results on Software SIMD. This work also operates only on positive, scaled data and provides more information on the scaling approach. They however have no good solution for multiplications and use very large guard bands whenever the algorithm requires a multiplication. [McK08] also uses a very wide guard band to pack two data streams of 5 bits into an 18-bit multiplier on an FPGA. This is still a nice example of exploiting the dynamic range, but the target of this implementation is to reduce the number of hardware multipliers of the FPGA that are used for a single application. The results are resented for an implementation of an FFT kernel and show also significant improvements in the FPGA power consumption, but the presented approach is much less flexible than the Software SIMD as introduced in this chapter.

In all the above approaches, the application is restricted to the emulation of homogeneous SIMD for powers of 2 on datapaths that do not support SIMD. This related work does not consider the link to the minimal word-width information from fixed point refinement and therefore assumes extra mask operations have to be inserted everywhere, which leads to smaller gains. Additionally, as no clear link is made to the range analysis part of the fixed point refinement, in many cases, more guard bits are needed in order to prevent overflows. None of the related work considers the packing of heterogeneous

or non-power of 2 word-widths and most importantly, they do not provide a solution for multiplications. As a result, the additional flexibility with respect to traditional Hardware SIMD is small, which restricts the applicability of the optimization to a small number of cases. Because of these restrictions, Software SIMD is mostly considered to be applicable only for a very low number of operations inside a kernel and the pack/unpack overhead is not decoupled from the potential of the parallelization technique.

A constructive approach has been presented that describes how to exploit fixed point refinement during parallelization, using Software SIMD. The technique differentiates between different types of operations and only adds extra corrective operations where needed. Arithmetic, logic and relational operations are supported, including saturation in software, for unsigned positive data. The overhead that is introduced by pack/unpack operations is estimated separately in order to motivate a modification of the data-layout where possible. As multiplications heavily reduce the efficiency of the Software SIMD technique, a conversion method is proposed to convert a large number of multiplications into other supported operations. This so-called strength reduction will be discussed in Chapter 10.

9.6 Conclusions and key messages of this chapter

Exploiting application information can have a large impact on the final performance or energy efficiency of the implemented system. This chapter has explored the potential of using word-width information during various mapping phases. In order to be able to quantify the effects of these optimizations, word-width aware energy models have been developed for datapath components. These models can also be used to steer optimizations that exploit word-width variation in the datapath. As the potential impact of word-width aware scheduling and assignment is restricted to the datapath logic only, which is often not the most important bottleneck, these optimizations are not explored further. Word-width aware parallelization, used as Software SIMD, has effects throughout the system and is therefore explored in detail, using two case studies. A first case study has explored the energy and performance effects of exploiting word-width information to increase parallelization for a simple sequential processor. A second case study performed a performance exploration across different data word-width for an 8-slot VLIW processor.

This chapter has extended the sensitivity of ISS-based energy estimation to cover variations in data word-widths. This extension allows designers to estimate the impact of various word-width aware optimizations on their system and to steer optimizations that specifically target energy reduction in the

datapath logic. The improved sensitivity of these models was demonstrated and has been proven essential to correctly evaluate the expected gains and steer the optimizations (20% difference with non-word-width-aware conventional approach). Word-width aware energy models can be generated using the presented flow. The models can be coupled to existing Instruction Set Simulators to generate fast and detailed estimates.

This chapter has shown the value of exploiting word-width information of application data during application mapping. A systematic overview was presented, covering all relevant compilation phases, from the SIMD/Vectorisation step (introducing the novel SoftSIMD approach) that decides on the parallelization down to the instruction selection, scheduling and assignment. The concept of these optimizations was described and the expected gains have been estimated. For instance, when applying the SoftSIMD for a MIMO kernel, a reduction in energy consumption and an improvement in performance of about 80%, when compared to using no SIMD, or of over 30% when comparing to Hardware SIMD has been demonstrated.

Software SIMD is much more promising and can deliver significant improvements over state-of-the-art Hardware SIMD implementations, due to its larger flexibility in handling different and even heterogeneous word-widths. However, if the algorithm contains multiplications, the applicability is compromised. Therefore, a conversion is proposed in Chapter 10. When deciding to apply Software SIMD, the overhead of a more complex pack/unpack (with respect to Hardware SIMD) and the additional overhead of extra operations that are needed to prevent the overlap of data in neighboring sub-words, should be taken into account. Additionally, a trade-off exists between repacking between different program phases (system scenarios [Ghe09]) and using wider data word-widths.

The most important restriction to be able to apply this work, is the availability of the required level of detail in the word-width information. If no information is passed on, nothing can be exploited. Active research in this area is looking promising. Partly automatic flows for data type refinement and fixed point refinement have already been proposed [Nov09, Roc06, Wid95].

CHAPTER 10

Strength Reduction of Multipliers

Abstract

This chapter presents a conversion technique for constant multiplications. The targeted multiplication operations (or MULs), which form a significant part of all MULs, are converted into (a number of) less complex, cheaper operations. Multiplier strength reduction is a well-known technique in hardware synthesis and has been used extensively for filter design in custom hardware. However, the specific context of embedded processors presents opportunities and trade-offs that have not been exploited before. The presented strength reduction significantly extends current multiplier strength reduction in compilers and can depend on a cost function in order to optimize performance, area, energy consumption and even operator accuracy, according to the requirements of a specific context. This approach enables the conversion of many more multiplications than are currently converted by state-of-the-art techniques. One of the main ways to exploit this transformation has been described in Chapter 9 where the SoftSIMD concept strongly benefits from the conversion of hardware multiplications into shift-and-add operations.

F. Catthoor et al., *Ultra-Low Energy Domain-Specific Instruction-Set Processors*,
DOI 10.1007/978-90-481-9528-2_10,

10.1 Multiplier strength reduction: Motivation

Use-cases for multiplier strength reduction Multiplier strength reduction, if it is approached as a general problem, can be used to serve different goals. Depending on the specific goal, the requirements and conversion trade-offs change. The following paragraphs describe different instantiations of the general conversion problem. This general scope, together with the systematic representation and exploration of the full search space, still within the context of embedded processor implementations, sets apart this work from the related work.

Firstly, the MUL-conversion can be used as a stand-alone technique to improve either the performance or the energy efficiency of an application or kernel, or even to reduce the area of a processor by reducing the number of multipliers. A speed-up can be achieved for kernels that have a multiplication-related bottleneck, by converting the MUL operations into, e.g. ALU operations. The ALUs are typically a less scarce resource, thereby improving the compiler freedom and reducing the schedule length. Processors that are designed to achieve a high performance for these kernels often provide many parallel MUL-capable PEs (e.g. [Bou08]), which is very costly in terms of area. After performing a strength reduction on (part of) the multiplications, the number of PEs that can execute a MUL can be reduced. As the energy consumption of a multiplier is much larger than that of an ALU operation, the conversion can also directly lead to energy savings. Examples of all three types of improvements will be given in Section 10.4.

Secondly, the MUL-conversion can be one example of a word-width aware ISA selection optimization when the word-width information would be used to select one of various mappings to (a set of) operations, as was shown in Figure 9.12. If the application domain contains a large set of constant multiplications with small factors (not necessarily with constants), a cheap multiplier implementation with restricted width can be specifically added to handle those efficiently. Based on the word-width information of the constant, the compiler can select this FU over a more costly one.

Thirdly, this technique can be used as an enabling step for optimizations that can only be applied to multiplier-free code (or for which multiplications are too expensive to handle), like the Software SIMD that was presented in Chapter 9. By replacing the MUL operation with other, less complex operations, more freedom is generated for other optimizations and improvements on performance or energy efficiency can be achieved. It is essential to recognize the enabling character of the conversion, as in that case the MUL-conversion should even be applied in cases where e.g. a performance penalty is realized, as long as the following optimization will move the design back to a more optimal point. This aspect is not covered in the related work.

The applicability of the work presented in this chapter as an enabler for Software SIMD was demonstrated in Section 9.4.4. All multiplications from the FFT kernel are successfully converted into ALU operations.

One more aspect that can heavily influence the conversion cost is the implemented operator accuracy. When the decision is made to convert a single MUL into a series of cheaper operations, the natural opportunity arises to limit the number of post-conversion operations. If the application accuracy requirements are taken into account, the inserted error that is the result of this restriction can be acceptable. These optimizations can lead to significant area and energy reductions.

The rest of this chapter is organized as follows. Firstly, an overview is presented of the different types of multiplications and which part is targeted in this work, together with a motivational example for strength reduction in the context of performance improvements (Section 10.2). Then the global conversion space is systematically covered (Section 10.3). This space covers all above discussed contexts. The experimental section of this chapter (Section 10.4) focuses on using the MUL conversion as a standalone technique for a set of relevant applications. Different applications require different conversion strategies and examples are presented of both exact conversions, as well as conversions in combination with a trade-off on operator accuracy.

10.2 Constant multiplications: A relevant sub-set

Even though the replacement of a multiplication with multiple other operations seems to have a negative effect on both performance and energy efficiency, often the opposite can be achieved. This section will motivate how to use the MUL-conversion to improve performance on parallel embedded processors. The case for energy reduction is similar but even more opportunities are present there when also distributed loop buffers (see Section 3.6.4) and advanced power gating are used. Parallel embedded processors provide a set of parallel Processing Elements or PEs to reach the required performance. In order to be energy and area efficient, most processors are heterogeneous: not all PEs provide the same functionality and each contains a set of FUs, depending on the application domain. PEs that support multiplications (MULs) are $expensive$, in cycles, energy and availability (they can be a bottleneck resource). State-of-the-art VLIW processors that can execute eight operations in parallel, most often only support two multiplications to be started in a single cycle, while they support up to 8 add-based operations in parallel (e.g. most of the TI processors do [TI06, TI00, TI09c]). During the execution of compute intensive kernels, this can become a severe bottleneck.

Strength reduction for constant multiplications can convert a subset of these MULs into less expensive operations. This chapter describes the *complete conversion space* and a technique for *strength reduction of multiplications by an integer* emphconstant at compile time. By converting these multiplications into a series of equivalent cheaper operations, the bottleneck can be removed. More specifically, a conversion into shift and add operations will be used to explain the different conversion techniques. Other target operations are possible however and will be discussed as part of the conversion space. Some optimized results will be shown in the experimental Section 10.4. A large amount of related work exists on the strength reduction of multiplications by an integer constant and a detailed comparison with that work will be presented in Section 10.5.

Current state-of-the-art deals with only a subset of all possible conversion methods, uses only performance as a cost-function and, most importantly, does not consider the underlying specific requirements and opportunities of the embedded processor context, e.g. the VLIW instruction set. This chapter does take those requirements into account and presents a systematic description of the complete exploration space. The presented conversion technique uses a heuristic in combination with multiple decomposition methods to obtain the cheapest conversion with respect to a cost function (e.g. performance). Instruction set extensions are proposed to obtain more efficient conversions.

The results show that for many applications a high percentage of multiplications can be converted. The introduced technique achieves a significant strength reduction in almost all cases. The technique achieves a speedup of 15% for a motion compensation kernel (MC) from the mpeg2 decoder, which translates to a 6% performance improvement for the complete application. Trading off accuracy to implementation cost by taking into account the application accuracy requirements can lead to larger gains. This aspect can be coupled to the processor architecture exploration, where an over-all reduction of the number of application multiplications can lead to a reduction in the number of hardware multipliers that need to be provided in the processor. When the hardware that is freed by this reduction can be used for additional ALU slots, performance increases of up to 30% can be demonstrated, as shown in Section 10.4.

10.2.1 Types of multiplications

In the compute intensive kernels of heavily optimized embedded applications, multiplications are often competing for resources and can be a bottleneck while other resources are under-utilized. However, many multiplications are in fact constant multiplications, which can be converted at compile time. Traditionally, these multiplications have been a target for removal by compilers.

In general, multiplications can be of five types.

- Type 1 between two variables ($var1 * var2$). This type can not be decomposed at compile time and is not the target of this work. Especially in a wireless communications context, many multiplications that are seemingly of this type can be traced back to a set of constant parameters that change depending on the modulation scheme or coding rate. However, using techniques like scenario's [Man10], these MULs that are currently considered to be variable, can be treated as constant multiplications and move to the other categories.
- Types 2, 3 and 4, namely $cons1 * cons2$ ($cons$ for constant), $induction\ variable * cons$ and $var * 2^n$, are completely known at compile time and will be expanded by state-of-the-art compilation techniques. Therefore, these multiplications are considered to be already removed and the results that are shown in the rest of this chapter do not include multiplications of types 2, 3 and 4.
- Type 5, $var * cons$ MULs, can be broken down, e.g. into a sequence of shift and add operations. However, the standard MUL-conversion can be done using a range of techniques of which no single technique is always better than the others.

Figure 10.1 shows that the percentage of MULs of type 5 to the total number is $significant$ for a representative set of benchmarks from $spec$ (Standard Performance Evaluation Corporation), $mediab$ (Mediabench) and our $internal$ benchmarks from the multimedia and wireless communication domains. This experiment was conducted using the COFFEE Tools [Rag08b], which includes the conversion mentioned above. The results show the ratio after the multiplications of types 2, 3 and 4 have been removed by the compiler.

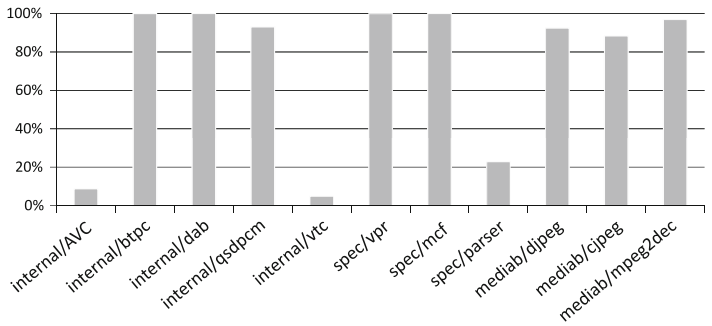


Figure 10.1: Ratio of constant MULs (of type 5) to total number of MULs (type 1 + type 5), counted during execution (dynamic)

Most existing conversions start from a specific context or optimization goal and propose a conversion based on fixed criteria, intended for a specific context (e.g. MULs in digital filters for ASICs). This leads to rather rigid restrictions and heuristics (e.g. only convert if the total number of inserted new operations is less than 3), which leads to a rather low number of effective conversions. Additionally, these approaches do not exploit the trade-off opportunities between performance, energy and other metrics that might be important in the embedded systems context.

Aspects of the conversion, or that influence the conversion, which are missing in the state-of-the-art, are added to the conversion space. Examples are given of how they can potentially improve the conversion. More specifically, three new aspects are added. Firstly, the link with the underlying processor implementation is taken into account explicitly, as this heavily influences the conversion cost. The instruction set can be extended, making this technique an input to the architecture exploration. Secondly, a method to trade off the implementation cost of the MUL with the accuracy is added, which allows to reduce the number of operations in case the application can tolerate the introduced error. Finally, a cost-function driven conversion is proposed to give designers control over the involved energy-performance-area trade-offs and potentially push the conversion technique such that all MULs in a certain kernel are converted, even at an extra cost, if this would enable additional optimizations afterward.

In conclusion, this section presents a technique for strength reduction of MULs of type 5 at compile time: expensive constant MUL operations are replaced with cheaper/weaker ALU operations.[1] A systematic description of the complete relevant conversion space enables this work to differentiate with respect to the state-of-the-art in the following aspects:

- Use of a heuristic-based search that takes the platform costs and optimization targets into account (e.g. special ISA-extensions, the use of the conversion as an enabling step, minimize register file accesses).
- Enable a trade-off between implementation cost and operator accuracy.
- Link to architecture exploration through the reduction of the required number of multiplier slots and ISA extensions.

As a result, many more (or different) multiplications can be converted than are currently converted by state-of-the-art techniques.

[1] The results shown in this work consider an operation to be cheaper/weaker if more PEs support this type of operation (defined in Section 10.4). This definition is for illustrative purposes only. This definition can be extended to include a weighted function of energy, performance and area or any other metric.

10.2.2 Motivating example

Strength reduction [Muc97] is a well-known compiler technique which replaces costly operations with cheaper ones. Various types of strength reduction for constant multiplication exist.

Most state-of-the-art compilers replace multiplications of types 2, 3 and 4 (as introduced in Section 10.2.1) and a subset of type 5 MULs that are easy to convert (e.g. $(2^n + 2^m)$ or MULs with small constants that convert into less than three shifts and adds). After removing these, Figure 10.1 shows that the number of MULs type 5 is still significant for a large number of benchmarks. While in some benchmarks the ratio of constant MULs to all MULs is low, they are often part of kernels and can still form a bottleneck. Therefore, more advanced strength reduction techniques can be considered to reduce this type of multiplications. These techniques however should not only be more advanced in the conversion they perform, but also in selecting the best conversion strategy for the specific context. Which conversion method to apply or even if the conversion should be applied at all (in some cases the multiplier is still the best choice), can heavily depend on the optimization target (e.g. improve performance, even at the cost of an increase in energy) or the type of processor (e.g. very parallel, with many PEs and some space in the schedule to accommodate extra operations vs. not so parallel).

Typically, an 8-issue VLIW processor, as is shown in Figure 10.2a, contains only two slots that allow multiplication (e.g. slot 1 and 5), while an ALU operation can be performed in any slot. This is similar to the .M1 and .M2 of the TIC320C64x [TI00]. These *expensive* multiply slots can easily become the bottleneck while executing kernels. One such schedule is shown in Figure 10.2b, showing ALU operations as [1–8] and MULs as [A–D]. The schedule contains MUL operations of two types: between one variable and one constant (A and C, can be handled by our conversion) and between two variables (B and D cannot be removed at compile time).

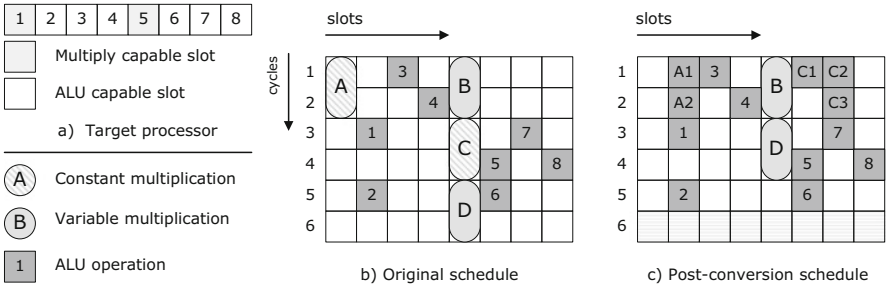


Figure 10.2: Example of the effect of strength reduction and slot availability on performance

Figure 10.2c shows a schedule where A and C have been converted. Since the inserted ALU operations (A1, A2 for A and C1, C2, C3 for C) can be executed in any slot, previously unused and less scarce resources can be used. The multiplication is no longer the bottleneck and the denser schedule leads to an improved performance (the schedule length is reduced by one cycle). Current compilers only support a fixed conversion that does not allow to explore this opportunity. This chapter shows that, by exploring the complete search space, an efficient conversion can be performed (see Section 10.3).

As part of this motivating example, a flat conversion (all constant multiplications are converted, unconditionally) has been performed for all applications of Figure 10.1. This experiment has been performed using the COFFEE compiler, which has been modified to detect the constant multiplies and appropriately report the required number of operations required after conversion (details on the conversion will be presented in Section 10.3).

Figure 10.3 shows the average number of operations inserted per constant multiplication per application, comparing the code that is using multipliers ($original$) with the result of a conversion to normal shift and add instructions ($with\ ALU$) and two types of special operations which allow a more efficient conversion (SA and SSA respectively). More details on the different ISA extension possibilities are discussed in Section 10.3.4. The plot has been normalized to the number of constant multiplications in the corresponding benchmark. It can be seen that in most cases the conversion of the constant multiplication resulted in inserting two to four operations. In case of cjpeg and djpeg, it was observed that the constants that are used for the multiplication are large and therefore the number of operations that are needed is

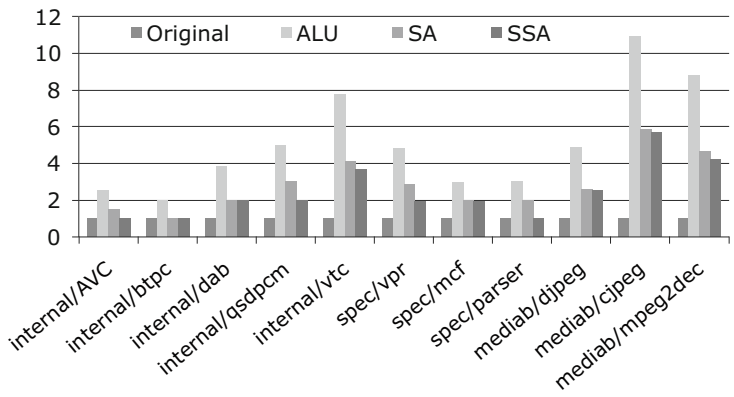


Figure 10.3: Average number of new operations inserted after the conversion, per MUL (original), using different specialized instruction variants (standard ALU vs. SA vs. SSA)

also large. In such cases, it may be desirable to still use the multiplication, as a few multiplier slots will still be available. This decision should be steered by the cost function.

As the number of inserted operations does not take into account the number of slots that support that specific operation, a more abstract $strength$ metric has been defined as follows:

Resource-aware Strength The resource-aware strength, called strength from here on, of an operation is defined here as the reciprocal of the number of functional units that can perform the operation. This definition of strength[2] is an area/performance oriented criterion, used here in the context of parallel processors and performance improvement. Other criteria can be defined specifically for performance, based on the critical path or for energy, based on the energy cost of the operations. In case two slots of the VLIW can perform multiplications, the strength of a multiply operation is equal to $1/2 = 0.5$. The strength of an ALU operation is then $1/8 = 0.125$.

Figure 10.4 shows the effective strength reduction that was achieved using a flat conversion. The graph is normalized to the strength of all constant multiplications for the corresponding benchmark. It is assumed that the SA and SSA operations can be executed in any of the eight slots, which is reasonable, given the limited area overhead, as is detailed in Section 10.3.4. It can be seen from Figure 10.4 that in some cases (5 out of 11) the strength of the multiplication is already reduced when an ALU operation is used.

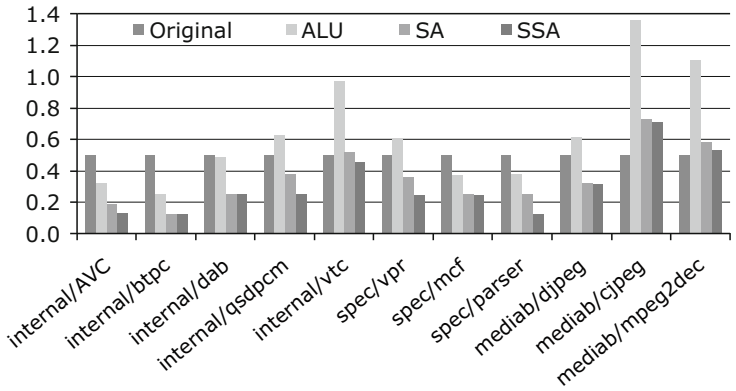


Figure 10.4: Resulting strength reduction for different specialized instruction variants for a brute force conversion

[2]In literature, no consistent definition for $strength$ can be found, due to the context specific use of strength reduction in general. The consensus is that strength reduction is a technique that improves compiler-generated code by reformulating certain costly computations in terms of less expensive ones. Therefore the definition of strength depends on the optimization criteria that define the cost.

The conversion is done in a brute force way. Therefore, the net strength after conversion can be higher (cjpeg, mpeg2dec). In case the extended instruction set is used, a significant reduction in the strength can be achieved in almost all the cases (9 out of 11). On average over all the benchmarks, the strength is reduced by 30% after converting constant MULs to ALU operations.

Overall, the positive effect on performance of this conversion depends on two aspects. Firstly, the multiplication should be a bottleneck resource and secondly, free space should exist in the schedule to accommodate the inserted operations. More detailed performance results are presented in Section 10.4, but first the conversion space is introduced.

10.3 Systematic description of the global exploration/conversion space

The complete relevant conversion space is described in this section, with respect to six different aspects, represented by the tree in Figure 10.5. The first branch contains $Primitive\ Methods$, which are techniques that completely reduce any given constant multiplication directly into a sequence of operations of lower complexity. A second branch, $Partial\ Methods$, contains methods that reduce the given constant into smaller constants of which some potentially can be converted more efficiently. To further convert the smaller factors into a series of less complex operations, a primitive technique has to be used. The $Coding$ branch indicates various encoding schemes or recodings that can be considered for representing the constant. Depending on the selected coding, the primitive methods will produce a different conversion with a corresponding difference in cost. The fourth branch covers the relation to the platform: the $ISA\ Extensions$ branch specifies the operations that are available on the platform. In the following text, it is assumed that the default case for this tree is the conversion to a sequence of shifts and adds. Potentially, more complex operations can be added to improve the conversion efficiency. The $Optimization\ Techniques$ branch contains some transformations that can be applied to a conversion. The final branch covers the $Implementation\ Cost\ versus\ Accuracy\ Trade-off$, the modification of the exact constant value in order to reduce the conversion cost.

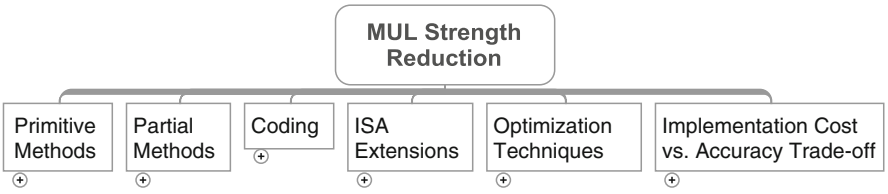


Figure 10.5: Overview of the constant MUL conversion space

Note that any branch can be combined with any other, and the final shift/add decomposition is only fixed after deciding on all six aspects. The conversion tree describes the complete conversion space. We propose to use a search strategy to cover this space, and to select the cheapest conversion given a cost function (e.g. performance, energy, etc.).

10.3.1 Primitive conversion methods

Primitive methods systematically convert a constant into a sequence of shifts and adds (see Figure 10.6). Two main approaches can be identified based on the parallelism of the produced conversion. The first method, called $Bitwise$, is the most parallel one and generates shifts for every separate non-zero bit of the constant based on the position of that bit in the word. Those intermediate results are then combined into the multiplication results using adds (two by two). The second method, called $Recursive$, generates the shifts in an iterative way based on the relative position of the non-zero bits with respect to other non-zero bits. In this approach, every shift starts from the previous intermediate result, which leads to a more sequential conversion. Hybrids between these two methods can be formed. represented as a binary word (or an other coding), which can be the complete constant, or the result of a partial method.

The examples presented here apply left or up shifts. However, a similar approach can be followed by using right or down shifts. An example of this will be given in Section 10.3.6 to illustrate the impact of this choice on the accuracy of the results.

10.3.1.1 Bitwise (or parallel) method

The $Bitwise$ method (see Algorithm 2) inserts shift operations that shift the $variable$ over a number of positions, indicated by the $shift\ vector$

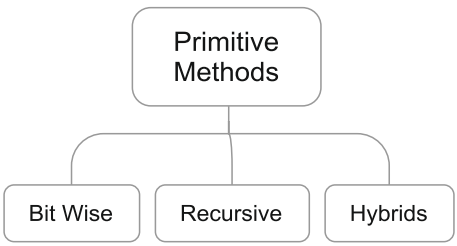


Figure 10.6: Primitive conversions convert the input constant completely into a set of target operations

Algorithm 2 Bitwise Method

Input: Multiplication: $var \times constant$
{Move over bits of constant from LSB → MSB side}
1: $pos \Leftarrow 1$ {Keep track of position of bit}
2: **while** bits left in constant **do**
3: **if** $bit(pos) = 1$ **then**
4: $var \ll pos$
{Insert Shift operation}
5: $pos{+}{+}$
6: Continue to next bit
7: **else**
8: $pos{+}{+}$
9: Continue to next bit
10: **end if**
11: **end while**
12: **for all** Inserted Shift operations **do**
13: Add pairwise to get final result
{Insert Add operations}
14: **end for**
15: **return** Sequence of Shifts and Adds

(sh1, sh2, etc. in Figure 10.11), that corresponds to the position of that respective non-zero bit, moving over the $constant$ from the LSB (Least Significant Bit) side to the MSB (Most Significant bit) side. The partial results are pairwise added, until the final result is obtained.

This method typically results in rather large shift vectors (of maximum size equal to the word-width of the data, e.g. 32), but all these shifts can be performed in parallel (depending on resources). The minimal number of cycles needed for a constant with N non-zero bits is $(\lfloor log_2(N-1) \rfloor + 2)$ cycles. The $parallel$ nature of this method (see Figure 10.11a) produces good results when the cost-function tries to optimize performance (minimize latency).

10.3.1.2 Recursive (or sequential) method

The $Recursive$ method (see Algorithm 3) moves over the bits of the constant from MSB to LSB and (except for the first non-zero bit) inserts a shift every time a non-zero bit is found, with a shift vector equal to the number of bits between the previous and the current non-zero bit, plus one. The $variable$ is added to the result of the previous shift operation and then used as input for the next shift. This method produces a $sequential$ series of operations (see Figure 10.11b), but the shift vectors are on average smaller than those produced by the bitwise method (although no direct guarantee can be provided on the maximal size). Therefore, this method is better suited for VLIW processors that support cheap small shift vectors and hence can be more area and energy efficient (e.g. due to reduced loop-buffer requirements). As the

Algorithm 3 Recursive Method

Input: Multiplication: $var \times constant$
{Move over bits of constant from MSB → LSB}
while $bit = 0$ **do**
  Continue to next bit
end while{Reached first non-zero bit}
$count \Leftarrow 1$ {Count number of 0-bits}
$temp \Leftarrow var$ {Keep intermediate result}
Continue to next bit {Skip first 1-bit}
while bits left in constant **do**
  if $bit = 0$ **then**
    $count{+}{+}$
    Continue to next bit
  else
    $temp \Leftarrow (temp \ll count) + var$
    {Insert Shift and Add operations}
    $count \Leftarrow 1$ {Reset counter}
    Continue to next bit
  end if
end while
if $count > 1$ && $LSB = 0$ **then**
  temp = temp << (count-1)
end if
return Sequence of Shifts and Adds

partial results of every shift-add pair are used as the input for the next shift, the minimal number of cycles needed for a constant with N non-zero bits is $2N - 2$ ($2N - 1$ if the LSB bit $= 0$).

Although there is a clear difference in the minimal number of cycles that is required for both primitive methods ($O(\log_2(N))$ for Bitwise and $O(N)$ for the Recursive approach), the number of operations that is required is equal for both methods, namely N Shifts and $N-1$ Adds. In case the most LSB-side bit is non-zero, one Shift less is required, leading to a total of $N-1$ Shifts and $N-1$ Adds. In both cases, the number of operations follows $O(N)$. However, the style of shifts is different (as mentioned above), with smaller shift vectors for the Recursive method.

The primitive methods generate different conversions, depending on the coding that is used (see Section 10.3.3). The final cost depends on the platform ISA (see Section 10.3.4). So, all combinations of these branches can be considered.

10.3.2 Partial conversion methods

A second branch of the search space is formed by different methods that handle parts of constants efficiently, represented by Figure 10.7. The splitting, here called $Factoring$, can be done following two approaches:

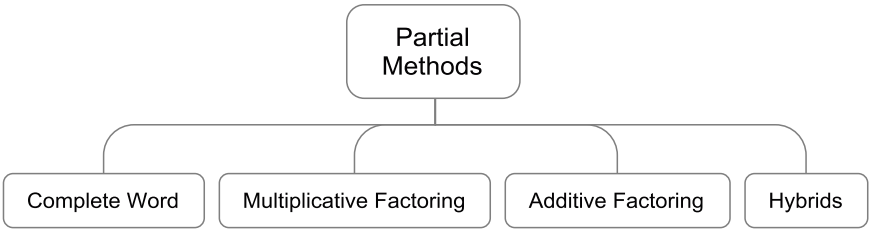


Figure 10.7: Partial methods split the input into a set of less complex constants

$Multiplicative\ Factoring$ or $Additive\ Factoring$ (explained below). Once the split is performed, each $factor$ can be handled separately using a primitive method. A partial method is beneficial if the cumulative cost in terms of energy or cycles, needed for the conversion of all factors, is smaller than for a primitive method. This subsection discusses the only two available types of factoring that can be used for mainstream arithmetic, namely multiplicative and additive factoring. Also, in this case, hybrid approaches are possible, by recursively using different split decisions on a constant.

10.3.2.1 Multiplicative factoring

In multiplicative factoring (which corresponds to traditional factoring), the constant is broken down into its factors ($const = cons_1 * cons_2 * \cdots * cons_n$). Potential candidate factors can be found by first splitting the constant into prime factors (see Algorithm 4). As the final cost of the conversion needs to be minimal, it is beneficial to look for larger $good\ factors$ first. Good factors are defined as factors that can be converted efficiently. Therefore, combinations are generated using the prime factors, and sorted in descending order. They are converted if they are of the form (2^n) or $(2^n \pm 1)$, which leads to a single shift or a shift and an add only. Other $good\ factors$ that can be selected are $(2^m \pm 2^n)$, two shifts and one add, and so on. Which good factors to look for and how they are preferred or discouraged by the cost-function depends on the underlying VLIW instruction set (see Section 10.3.4).

Consider the example of Figure 10.8: $var * 45$. The binary representation of 45 is 101101, which leads to three shifts and three adds, when directly using a primitive method ($N = 4$). However, when 45 is factored into $3 * 15$, a more efficient conversion is possible, using a temporary variable: $temp = [(var \ll 4) - var]$, because $15 = 2^4 - 1$, to first generate $var * 15$. This result, $temp$ can then be multiplied with 3, by doing $(temp \ll 1) + temp$, because $3x = 2x + x$. This takes only two shifts and two adds (subtract) in total. However, the minimal depth of the dependency tree has increased from

Algorithm 4 Multiplicative Factoring

Input: Multiplication: $var \times constant$
1: Break down $constant$ into prime factors
2: Sort factors in ascending order
3: $temp \Leftarrow var$ {Keep intermediate result}
4: $twocount \Leftarrow 0$
5: **while** $current\_factor = 2$ **do**
6: $twocount{+}{+}$
7: Remove 2 from list of factors
8: Go to next factor
9: **end while**
10: $temp \Leftarrow (temp \ll twocount)$
{Insert Shift for factor $2^{twocount}$}
11: $factor[n]$ {array of remaining factors, n in total}
12: **for** $i = n$ downto 1 **do**
13: $comb[k]$ = set of C_n^i of $factor[:]$
{$comb[k]$ is array of combinations, k in total}
14: Sort $comb[k]$ in descending order
15: **for** $j = 0$ to k **do**
16: **if** $comb[j]$ is of form $(2^x + 1)$ **or** $(2^x - 1)$ **then**
17: Remove contributing i factors from list
18: $temp \Leftarrow (temp \ll x) + temp$ **or**
$temp \Leftarrow (temp \ll x) - temp$ resp.
{Insert Shift and Add (Subtract)}
19: **goto** 11
20: **else if** $Test\ for\ more\ interesting\ forms\ here$ **then**
21: Insert corresponding Shift and Add
22: **end if**
23: **end for**
24: **end for**{No more $good\ combinations$ found}
25: **for all** $factor[k]$ with $k = 1$ to $rem\_factors$ **do**
26: $min(Bitwise(temp, factor[k]),$
$RecursiveMethod(temp, factor(j)))$
27: Insert corresponding number of Shifts and Adds
28: **end for**
29: **return** Sequence of Shifts and Adds

3 to 4 in Figure 10.8. The effect of this increase on the required number of cycles depends on the number of parallel slots in the architecture and on the use of techniques like software pipelining.

10.3.2.2 Additive factoring (word splitting)

Another possibility of splitting constants into parts is to split the binary word into different parts by selecting for every 1-bit into which part it is put, e.g. splitting 101101 into 101000 and 101. Interesting parts to handle separately are the targets for the optimization techniques of Section 10.3.5, e.g. repeated bit patterns, for which the conversion can be re-used. In general, more complicated splits can be made, e.g. 101101 into 100100 and 1001, in which the non-zero bits in both parts are in non-distinct sub-words.

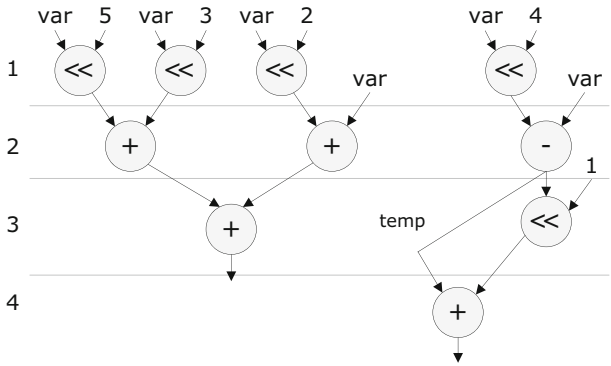


Figure 10.8: Comparison of two conversion options for $var * 45$, trading of the number of operations for the minimal latency

By identifying the interesting parts to split off, called $good\ additive\ factors$, the number of operations can be reduced. As with multiplicative factoring, which good factors to look for and how they are preferred or discouraged by the cost-function depends on the underlying VLIW instruction set (see Section 10.3.4). Therefore, the algorithm to exploit additive factoring is not described in general here, but the following classes can be targeted:

- Repeated bit patterns: Form a target for Common Sub-expression Elimination or CSE (as explained in Section 10.3.5)
- Long runs of 0's: Selectively reduce the size of the shift vectors used in the following conversion of the MSB part(s)
- Equidistant splits into parts: Uniformly reduce the size of all shift vectors to a size equal to largest supported shift distance in the hardware

The experiments presented in Section 10.4 only make use of additive factoring in the context of CSE.

10.3.3 Coding

A third aspect of the MUL-conversion methods is the coding that is used to represent the constant, represented by Figure 10.9. In ASIC designs, it is well-known that the choice of encoding has an impact on the implementation of multipliers and the number of shifts and adds required internally. In [Hew00], the usage of Canonical Sign Digit (CSD) representation is discussed. [Par01] introduces a Minimal Signed Digit (MSD) representation. In [Pat98], the authors discuss the usage of Booth encoding.

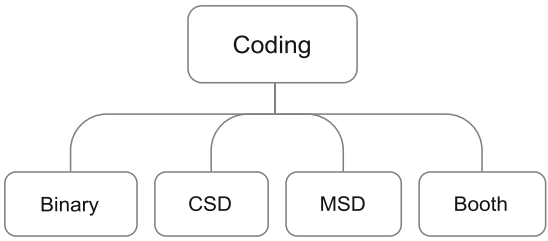


Figure 10.9: Different encodings result in a different number of non-zero bits and different optimization opportunities

In the context of processor implementations, this aspect directly translates to the number of operations that are needed after conversion. When a constant is represented by its binary coding, the conversion of the constant multiplication to shifts and adds requires a number of shifts equal to the number of non-zero bits (see the primitive techniques). By allowing subtractions (represented by $-$), fewer shifts are required. The CSD form is represented by using 1 for addition and $-$ (instead of -1) for subtraction. For example, the multiplication by 27 (11011 in binary), can be rewritten as $100-0-$, which corresponds to $((32)+(-4)+(-1)=27)$. This requires fewer shifts and adds/subtracts than the binary case. Re-coding the binary constant in to a CSD representation, uses the subtraction and at the same time guarantees that only the minimal possible number of non-zero bits is used. A CSD form is unique (a number has only one CSD representation), thanks to the additional restriction that no two neighboring bits are both non-zero. When this extra requirement is removed, but the number of non-zero bits is still minimal, the representation is called MSD (this is not a unique representation any more). In the context of this work, the MSD representation will normally be preferred as it provides more flexibility and reduces the total word-length (e.g. 11001 is one bit less than the CSD form $10-001$). In state-of-the-art compilers, CSD is often used as the coding of choice, as it leads to the smallest number of operations for complete conversion of the constant. In this work, the target is a conversion with a minimal cost, which also depends on the underlying VLIW hardware. Therefore, the trade-off is not that simple. The combination with partial methods could mean that e.g. the binary or MSD representation could give better opportunities for CSE. Therefore, all coding variants can be considered. Booth coding provides similar opportunities by handling pairs of bits together.

10.3.4 Modifying the instruction-set

The branch of Figure 10.10 shows some potential hardware implementations (FUs) for the multiplication operation. The Platform ISA tree is not

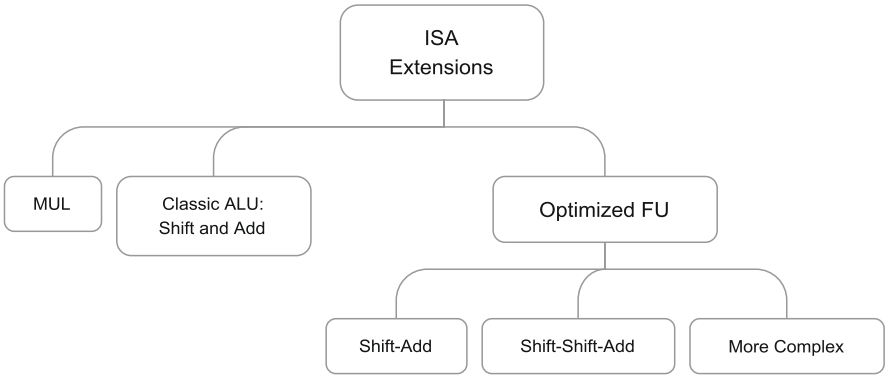


Figure 10.10: Different instruction-set extensions can reduce the conversion cost

fully systematically explored here, but we provide some example hardware options that are relevant to the embedded VLIW processor domain (e.g. using standard Shift and Add operations or combinations of those). The availability of special FUs heavily influences which conversion strategy should be followed, as it impacts the cost function when exploring the search space. The next section will describe some potential instruction set extensions that can improve the result of the conversion of constant MULs.

Instruction set extensions The strength reduction algorithms described in the previous sections produce sequences of shifts and adds. The produced sequence, represented by its data-flow graph or DFG, depends on the exact combination of primitive and partial techniques that was selected for a certain constant. This DFG still can be scheduled and assigned in different ways. Additionally, it is possible to modify/extend the instruction set of the architecture to better match the produced DFGs and in this way reduce the cost (e.g. improve the performance).

Given the case that no special FUs have been added to improve the efficiency of the conversion, the add and shift operations will be executed in the ALU unit (standard 32-bit ALU area=8,176 μm^2 for a UMC 90 nm Standard Cell technology). One of the recurring subgraphs that are formed using the recursive method is marked using a striped box in Figure 10.11b. Most instruction sets contain a single instruction which can execute a shift and add operation in a single cycle. As an ALU unit typically already consists of both the logic for the Add as for the Shift, providing a combined Shift-Add operation requires very little extra hardware. In the rest of the paper this functional unit is referred to as SA. The SA operation can be executed on the ALU. For the bitwise method, the common recurring subgraph is shown using a striped box in Figure 10.11a. This operation requires three inputs and pro-

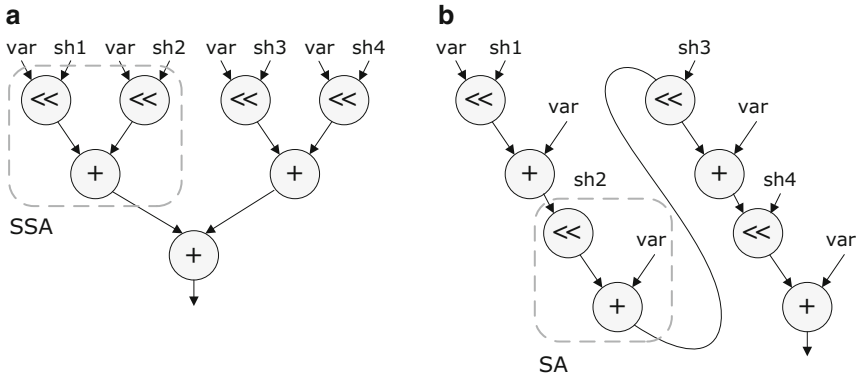


Figure 10.11: Bitwise and recursive method: The striped boxes show a recurring subgraph for both methods that can be implemented in a special FU, here called SSA and SA for (a) and (b) respectively

duces one output. The Shift vectors can be register allocated or encoded into the instruction. We refer to this functional unit as SSA (extended ALU, including SSA, area=10,620 µm^2, due to an extra shifter). The area numbers should be compared to the area of a 16-bit multiplier, which is 25,846 µm^2, which shows that providing the SSA is affordable, most importantly because most of the needed hardware is already available in the ALU and the extra overhead is limited. These numbers also indicate that, purely from an area perspective, it is not unreasonable to consider the possibility of adding extra ALU slots (in this case 2, including SSA support, can be adder per removed MUL) if the MUL-conversions can be used to reduce the number of MUL slots in the architecture (see the experiments in Section 10.4.4).

10.3.5 Optimization techniques

Some optimization techniques allow to efficiently convert constants or parts of constants and can be used together with the partial methods (see Figure 10.12). In general, the optimization techniques try to directly reduce the cost of the conversion by reducing the number of operations or indirectly, by modifying the DFG to match the ISA better, thereby reducing the number of instructions.

A common target for this type of manipulations is to apply $Common\ Subexpression\ Elimination\ (CSE)$. In this case, repeated bit-patterns in the constant can be identified and can be combined to form the full constant, e.g. in combination with additive factoring. One example is 101101, which can be split into 101 and 101, the common sub-expression. To get the original constant, the extra shift is needed to move the first part to its original

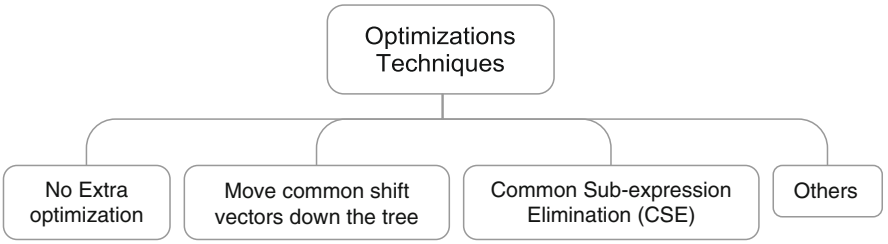


Figure 10.12: The total conversion cost can be reduced by applying optimizations across operations

position, 101000. For $var * 101$, one shift and one add are needed, using a primitive method. This result is then shifted over three positions, leading to $var * 101000$, to which $var * 101$ is added again, resulting in $var * 101101$, in a total of two shifts and two adds, instead of the six operations needed by directly using a primitive method.

Another interesting pattern to split off are $Runs\ of\ 1's$, which is actually the additive factoring equivalent of $(2^n - 1)$. This case is handled automatically when using a CSD encoding, but can be applied selectively in other encodings. For example when performing $var * 111111$, the 111111 can be converted to $1000000 - 1$, which leads to only one shift and one subtraction, instead of six shifts and five additions using the primitive methods.

Independent of the chosen conversion method, a combination of partial and primitive methods, the resulting DFG can also be manipulated, e.g. to match the available hardware. One example of this kind of optimization techniques is to move $common$ shift vectors down the conversion tree, as is shown in Figure 10.13

In principal, the full range of algebraic transformations (associativity, commutativity, etc.) are possible. A complete formal categorization of this space is part of future work.

10.3.6 Implementation cost vs. operator accuracy trade-off

This section discusses the trade-offs related to the accuracy that can be exploited when using constant multiplication strength reduction. Three aspects of this trade-off come into play here. Firstly, the controlled introduction of errors can be exploited to reduce the cost of the conversion. Secondly, different techniques can be used to control the width-expansion that can occur when multiplying. Thirdly, a difference exists between applying this for scalar data and the usage of the strength reduction in a Software SIMD (or other data parallel) context.

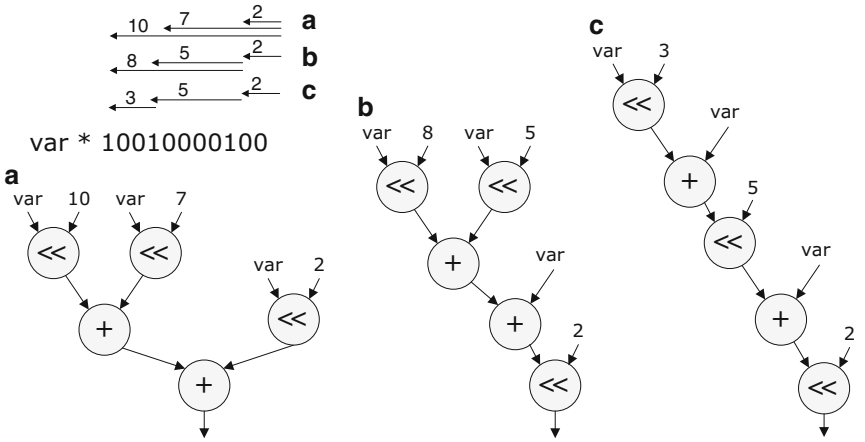


Figure 10.13: The DFG of the conversion tree can be transformed to match the underlying hardware, e.g. if a shifter that supports only small shift vectors is available. The Bitwise conversion of (a) is transformed to (b) by moving down the shift over two positions. The fully sequential version (c) is obtained by moving the common shift over five positions

10.3.6.1 Trading off accuracy with performance

The conversion of a multiplication into a sequence of e.g. shifts and adds can be used to trade off the operator implementation cost with the accuracy of the results. By matching the application requirements to the MUL-conversion, in many cases, the effective cost of the conversion can be reduced. Depending on the chosen technique and on how the conversion will be used, different costs and trade-offs come into play.

As the conversion methods that have been discussed in this chapter produce a sequence of operations for each converted multiplication, a natural way to reduce the cost is to restrict the number of operations used in the conversion and to truncate that sequence. Figure 10.14 shows an example of the Bitwise conversion (a), in which the smallest shift vectors are removed (b). This leads to an error on the produced product, but depending on the application requirements, this error can be tolerated in specific contexts. In the following example, a constant (773) is changed to another cheaper constant (768), because fewer non-zero bits lead to less operations after conversion.

Original MUL		Result of Truncation
$var * 773$		$var * 768$
binary	01100000101	binary 01100000000
CSD	10-00000101	CSD 10-00000000

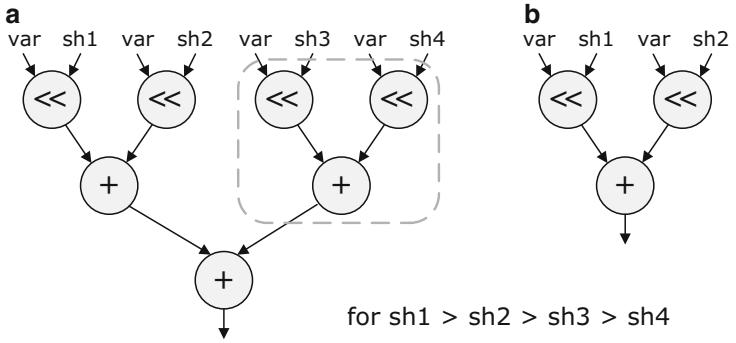


Figure 10.14: Accuracy vs. implementation cost trade-off, by restricting the number of shift operations. In this example, the smallest shift vectors are removed for a Bitwise conversion

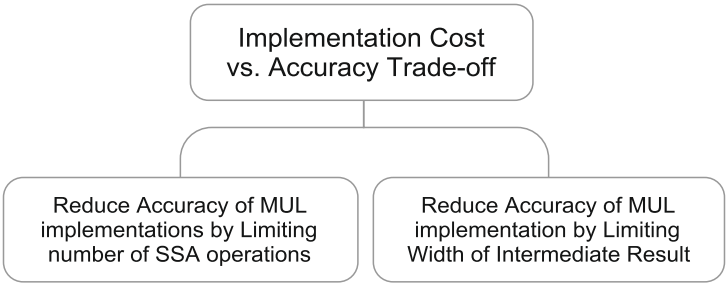


Figure 10.15: The implementations cost of a constant MUL can be traded off with the application performance

Alternatively, the constant can be modified directly (without first deciding on a conversion and then restricting the total number of operations) to a different $good$ constant, meaning it can be efficiently converted (see Figure 10.15). The difference with truncating the conversion tree, is that other techniques can be used to decide which other constant is interesting, e.g. based on CSD coding (a larger constant can be interesting) or CSE, which changes the conversion tree completely.

10.3.6.2 Preventing width expansion of multiplication results

Another aspect that affects the implementation cost of multiplications, is the decision on the word-width that is used internally and at the output of the MUL operation. This section will cover some possibilities and highlight the trade-offs in the context of the conversions.

Multiplication with a double width result In general, the result of the constant multiplication can be the sum of the widths of both input operands (e.g. a product of inputs of width w produces a result of width $2w$). In a realistic implementation, however, the width is not allowed to increase without restriction over time, so the result will be shifted back (mostly into the word-width of the inputs, as shown in Figure 10.16a). Therefore, some accuracy is lost when shifting back to width w.

Many DSP implementations support only multiplication that are half the width of the other operations, e.g. $w/2$ as shown in Figure 10.16b. In that case, the input operands are truncated and the cost of the multiplication is reduced, but an additional reduction in accuracy has to be taken into account.

In order to get the same result after strength reduction, the first possibility (for scalar operations) is to provide a functionality equivalent to version (a), which means the intermediate results are of increasing widths and shifting back is done afterward, as in case of the multiply. Figure 10.16c then corresponds to the use of SSA hardware that internally uses wider registers to allow the width expansion and produces a double width output. Note that the complexity is still much less than that of a multiplier, as the number of $partial\ products$ for the SSA operation is restricted to two only, while for a multiplier this scales with the word-width of the multiplicand. This corresponds to the fact that a multiplier can execute any multiplication of the supported width, while the SSA hardware only supports constant multiplications that can be converted into only two shifts and one add.

However, this approach (allow the double width internally) increases the cost of the SSA implementation and, in many cases, this extra cost is not needed, as the extra precision is not required and the accuracy reduction can

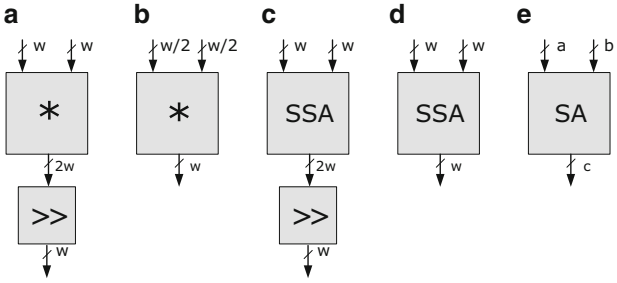


Figure 10.16: The word-width requirements of produced outputs, with respect to the inputs, and the implications for the accuracy depend on the implementation of the strength reduction. Different versions have different trade-offs with respect to their cost and usage restrictions

be tolerated by the application. This is especially true if the accuracy is traded off further to reduce the conversion cost, by restricting the inserted number of operations, as has been discussed above.

The increase in width can be avoided by propagating the shift that comes after the multiplication up using transformations, as shown in Section 10.3.5. As a result the output width of the SSA is immediately the required width and the extra shift is removed (Figure 10.16d). When using more parallel conversion methods, this can generate additional errors, because some bits will be shifted out of range before they can cause a carry. However, in many cases, this additional error can be tolerated and will still lead to results within spec. This is e.g. true for many applications from the wireless communication domain, as they are designed to cope with non-perfect channel conditions.

By taking the output width into account during the conversion and when using a sequential conversion strategy (all carry bits get the chance to propagate), the same results as for a multiplication can be obtained (Figure 10.16e, illustrated by an example in Figure 10.17), which will be explained below.

Strength reduction for Software SIMD Using a conversion strategy where the width of the intermediate results is allowed to increase will lead to problems in any data parallel approach where words are used that are a concatenation of sub-words, but even more so in a Software SIMD approach, as bits will eventually ripple to the neighboring sub-words. Therefore, extra care has to be given to the conversion in this context.

For Software SIMD, multiple solutions can be used to handle this problem. As a first possibility, as was used as a requirement throughout Section 9.4, the output requirements of the sub-words can be propagated to the input packing. This corresponds to the assumption that the word-width requirements of sub-words stay constant during the execution of a kernel (worst case or worst case during a phase when linked with scenarios). This approach, shown in Figure 10.16c, assumes that the instantaneous width of the input data is such that the result of the constant multiplications will fit into the word-width that has been assigned to each sub-word. As a result, the shifts of the normal sequential method will not move out the available word-width and no extra corrective operations are required. However, more flexibility in the Software SIMD packing can be supported by restricting the produced output width of the multiplication, as is normally done by shifting back the multiplication result into a smaller word-width. Figure 10.16e represents this solution, where the results of the multiplication of inputs with instantaneous data of widths a and b respectively can be restricted into an output of width c, with $c \leq a + b$.

Figure 10.17 shows an example conversion in which the multiplication of a 5-bit variable and constant produces a 10-bit result which will be cast back

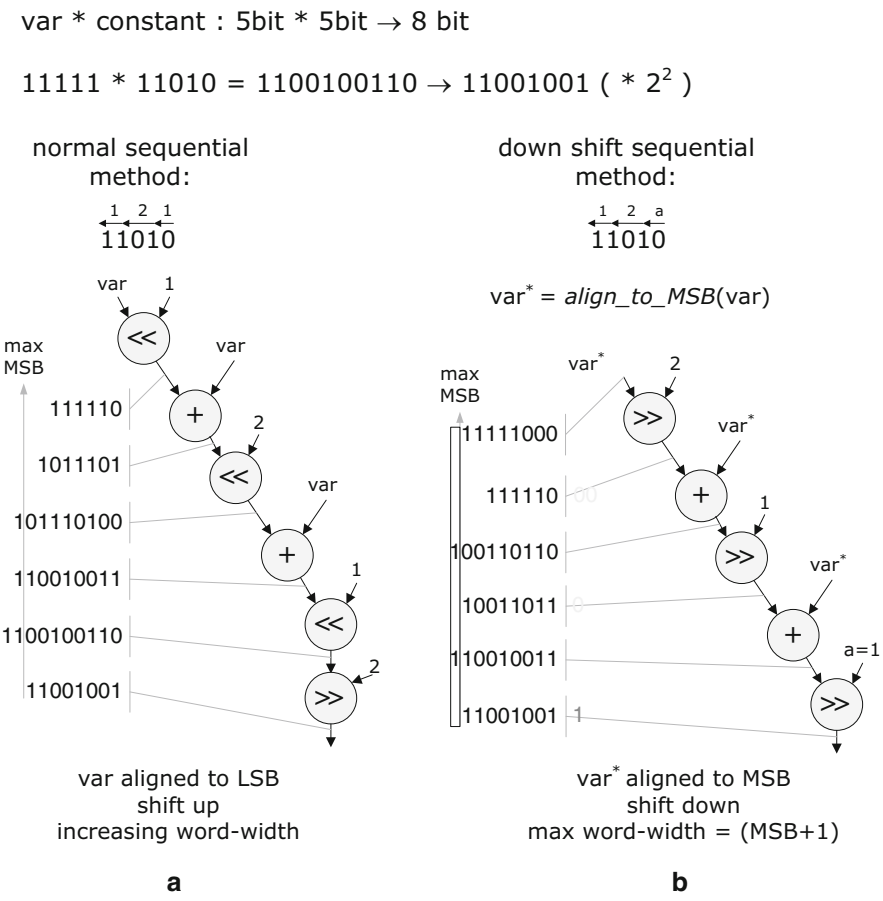


Figure 10.17: By aligning the data to the MSB side of the word and using a conversion based on right-shifts, the accuracy can be improved with respect to the input restriction method

into an 8-bit word (as shown at the top). Figure 10.17 a shows the sequential conversion method, which produces a sequence of shifts and adds in which each shift moves the previous partial product to the MSB side. The input to this conversion is the multiplicand variable, aligned to the LSB side. Some partial products exceed the width of the output (as indicated by the vertical line, max MSB). At the end of the conversion, the result is shifted back into the required width, causing an error only at the end. Using this method in a Software SIMD context is problematic, as the MSB bits will interfere with neighboring sub-words. Masking can be used to remove potentially offending bits, but in this case, removing the MSB bits will lead to large errors in the produced output. Figure 10.17b (an example of Figure 10.16e) shows an alternative implementation of the sequential method, here called the $down$

$shift\ sequential\ method$. In this case, the data is aligned to the MSB side (or to MSB-1 if no guard bit is used) first (var^*) and all subsequent shifts are down-shifts or shifts to the LSB side. As a result, the position of the MSB bit of the partial products is guaranteed to be smaller or equal to the required MSB increased with one (MSB + 1). To stop this one bit from causing interference with the neighboring sub-word, the same guard bit can be used as was discussed e.g. in the context of overflow detection. Note that the guard bit is not necessary here, as the initial alignment can be to MSB -1. At the start of this conversion, the position of the fractional point is changed by the re-alignment to the MSB side (e.g. by +3 positions in the example of Figure 10.17b). Subsequent shifts will modify this position again (e.g. -2 and -1, respectively in the same example). At a certain moment in time (compile-time analyzable), the number of positions from shifts to the LSB side can exceed the initial shift to the MSB side. At this point, LSB bits can interfere with MSB bits from neighboring sub-words. However, unlike in the normal sequential method, in this case, they can be easily masked (using a compile-time generated mask). In this case, LSB bits are removed during the subsequent right shifts with masks, instead of removing all redundant bits at the end only. As the down-shift sequential method allows carry bits from partial products to ripple and, as only LSB bits that exceed the specified output range are masked away, the result is equal to the multiplication result of Figure 10.16a. For some very small word-widths, still an error is possible, but this can be taken into account during the fixed point refinement and the mapping.

Instead of using a mask operation to remove the bits that are shifted out of the sub-word on the LSB side, a (temporary) repacking can be used to provide the extra bits. If a very flexible shuffler unit is available, the extra bits can potentially be shuffled out of the word. Which option is preferable depends on the respective costs and availability of these operations.

10.3.7 Cost-aware search over conversion space

As the conversion space that is covered by the trees described above is extremely large, heuristics are needed to steer the conversion or to be able to eventually automate this conversion and come up with an algorithm with reasonable compilation time. Finding an optimal set of heuristics to get the best results is outside the scope of this book. In the experiments that are presented in Section 10.4, the cost function aims to improve the performance. Therefore, a simple heuristic is used as a first example of a full conversion implementation for some realistic applications, targeting constant multiplications in a scalar context. The heuristic search starts by performing a $Bitwise$ or parallel conversion on the CSD representation of the constant. This approach

guarantees the minimal length of the dependency tree, assuming parallel execution because of the Bitwise conversion, together with a minimal number of operations after conversion, because of the CSD form. After this first conversion result, the search is used to find cheaper conversions, using factoring. A full search is performed on multiplicative factoring and a limited search on additive factoring, identifying only the interesting targets for the CSE (repetitive bit patterns). When a factoring approach is tried and the sum of all non-zero bits (using CSD) for different parts is not smaller than for the full constant, the search is stopped for this factoring. In this way, the search space is quickly reduced, and only the most promising paths are searched. As the end cost depends heavily on the operations that are supported by the platform, a high level cost metric $strength$ has been introduced to evaluate the success of the conversion (see Section 10.2.2). The strength metric takes the availability of operations on the platform into account. To show specific results for performance, the here described simple heuristic was used for a flat conversion (all constant multiplications have been converted).

Extensions to this simple heuristic should replace the flat conversion with a conditional conversion, still using a traditional multiplication for constant multiplications that can not be approximated by a cheaper constant and for which the exact conversion would require too many operations. In addition to this performance driven conversion, an energy-aware cost function is required. To enable this, energy estimates for the instruction set extensions, like SSA and SA, are required. The hardware design and detailed performance and energy estimations for such extensions are outside the scope of this book and are part of future work.

10.4 Experimental results

This section illustrates the strength reduction of multiplications for some practical examples. Firstly, the experimental procedure is explained in Section 10.4.1. A high level broad experiment shows the potential of a flat conversion method for a broad range of applications, using the abstract metric $strength$ (defined below). Sections 10.4.2, 10.4.3, 10.4.4 and 10.4.5 present more detailed performance estimates for conversions with different boundary constraints or optimization goals for a video application (an IDCT kernel, part of an MPEG2 decoder), a multi-antenna wireless communication application (FFT kernel, part of a MIMO application), a biomedical application (a DWT kernel, from a hearth-rate monitoring application) and a biotechnology application (detection kernel from an online monitoring application) respectively. Each section will discuss the implemented choice and the trade-offs and obtained results.

10.4.1 Experimental procedure

The proposed strength reduction algorithms of Section 10.3 have been used to convert the constant multiplications from a range of applications into ALU operations or more efficient SA or SSA operations. The conversion process has been performed according to the performance-improving heuristic of Section 10.3.7 and has been partly automated. A set of scripts have been implemented to perform the automatic conversion of constants using the primitive methods, the exhaustive search over all multiplicative factoring possibilities and over different coding variants. A manual step has been used to cover additive factoring and identify candidates for CSE. The original benchmark code has been modified manually to reflect the conversion and the resulting code has been compiled using the COFFEE framework [Rag08b].

As no accurate energy estimation numbers have been produced for the different ISA extension candidates, the results of this chapter focus on achieving performance gains as a first target, but rough energy estimates are presented for selected kernels to give an indication of the expected gains. The target architecture is a standard heterogeneous VLIW processor with eight slots, of which only slot 1 and 5 contain a multiplier.

10.4.2 IDCT kernel (part of MPEG2 decoder)

To show the effect of the strength reduction on a well defined cost metric, namely performance, the presented conversion technique has been applied on the mpeg2 decoder. The conversion is using the simple heuristic and the search over the trees of the conversion space, as has been discussed in Section 10.3.7. In order to show the effect of the conversion clearly, the bottleneck that is caused by the multipliers has been been exaggerated for the sake of argument, by assuming a multiplier latency of 6 cycles. Increasing the multiplier latency, in this case, simulates an application that is heavily bottlenecked by the multiplier. However, the experiment in Section 10.4.3 will show improvements for a more realistic latency of two cycles. The Motion Compensation kernel, one of the most important kernels of the application, contains multiplications with constants (a part of an IDCT, Inverse Discrete Cosine Transform) that are not converted by state-of-the-art techniques (the constants are relatively large and are not of one of the easy to convert forms, like $(2^m \pm 2^n)$). Using the presented technique, and using the special SSA operation (as introduced in Section 10.3.4), significant speedups can be realized for this application.

Figure 10.18 shows the schedules for the Motion Compensation kernel, respectively before conversion, after conversion into shift/add/subtract ALU operations, and after using the special SSA operation. The multiplications

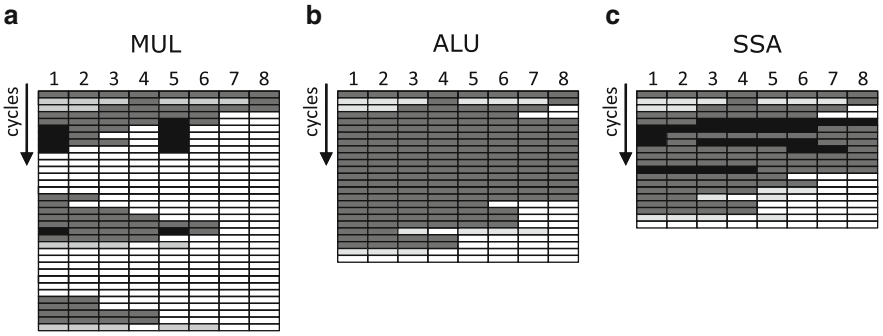


Figure 10.18: Strength reduction for the mpeg2 decoder before conversion (a), after conversion using standard ALU operations (b), and using special SSA operations (c). White: free slot; light grey: LD/ST operations; dark grey: ALU operations; black: MUL operations in (a) and SSA in (c)

are pipelined (a new operation can be started in the next cycle, even though they have a multi-cycle latency) and are therefore only marked in the cycle the execution starts. In Figure 10.18a, the constant multiplications (11 operations, shown in black) are forming a bottleneck, which is indicated by the completely empty lines in the schedule, due to their high latency. One iteration of this version of the IDCT kernel takes 35 cycles. The schedule still contains plenty of free space, which indicates that the cheaper ALU resources (gray operations in Figure 10.18a) are under-utilized. The multiplications are converted into multiple cheaper ALU operations in the second schedule (Figure 10.18b), which is much denser. However, it is often possible to schedule these in parallel, which leads to a speedup (only 24 cycles needed per iteration). When the special SSA operations are used (black operations in Figure 10.18c), the cycle count is reduced to 19 cycles per iteration.

As is shown in Figure 10.19, this leads to a performance improvement of about 45% for the version using ALU operations and 84% for the version with SSA support (normalized to the performance of the MUL version). This performance improvement for the IDCT kernel alone translates to a speedup of about 15% for the MPEG2 Motion Compensation and of 6% for the complete MPEG2 decoder application, by only optimizing the bottlenecked IDCT part of the Motion Compensation.

A first rough energy consumption estimate for the IDCT kernel alone (see Figure 10.19), using energy per activation estimates for a 40 nm TSCM standard cell flow[3], shows that the datapath energy for this kernel is reduced

[3]Cost of modeled operations per activation: MUL = 4,050 fJ, ALU op = 400 fJ, Shift on ALU = 200 fJ, Shift = 100 fJ, SSA = 600 fJ.

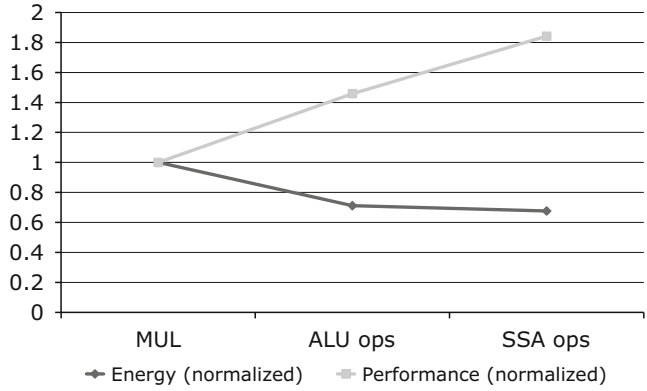


Figure 10.19: Impact of MUL-conversion on energy and performance of the IDCT kernel, normalized to the MUL version

with 29% when converting the MUL operations into ALU operations, while using the SSA unit leads to a reduction of 33% (both numbers normalized to the energy consumption of the MUL version). In this case, the performance improvement of the version with SSA support is even combined with a slight improvement in energy efficiency for the datapath operations. As the number of operations is reduced, taking into account a reduced number of accesses to the ICMO will lead to a larger energy improvement. This depends on the target platform and is not discussed here further.

For processors with a smaller multiplier latency (see also the following experiments) or in case the multiplier is not the main performance bottleneck, the conversion can then still be used to reduce the energy consumption, even at cost of some loss in performance.

In this experiment the constants of the IDCT have been converted using an exact conversion method (the conversion is I/O true). In many applications, however, accuracy requirements can be taken into account to reduce the conversion cost of larger constants. This is shown in the following experiment.

10.4.3 FFT kernel, including accuracy trade-offs

This experiment shows the conversion results for the FFT (Fast Fourier Transform) kernel, taken from the TI DSP library [TI09d] that also has been used in Section 9.4.4. The strength reduction is used there as an enabling step for the Software SIMD. In this section, however, the FFT kernel is used to illustrate the potential performance improvement when trading off implementation accuracy with operator cost. This is done by propagating

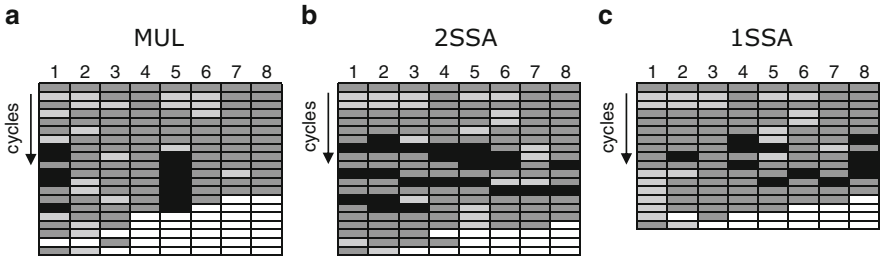


Figure 10.20: Strength reduction for FFT, before conversion (a), after conversion using 2 SSA operations per MUL (b), and 1 SSA operation per MUL (c). White: free slot; light grey: LD/ST operations; dark grey: ALU operations; black: MUL operations in (a) and SSA in (b) and (c)

application knowledge to the strength reduction step and taking into account the requirements to reduce the number of ALU or SSA operations that are used per replaced MUL operation.

Figure 10.20 a shows the original schedule, using MUL operations (in this case with a latency of 2 cycles). Using the conversion strategy described in Section 10.3.7, in this specific case, converting the constants required 2 SSA operations for an exact (I/O true) conversion. However, by reducing the maximum number of inserted SSA operations, the conversion cost can be reduced, at the cost of an accuracy reduction. Depending on the application requirements, the introduced error can in some cases be tolerated.

Figure 10.20 b shows the schedule for a conversion using 2 SSA operations per MUL, which leads to an energy gain for the datapath operations of 27%, without affecting performance (see Figure 10.21). Figure 10.20c goes one step further, by restricting the inserted number of SSA operations per MUL to 1. In this case, the performance increases with 17% and the energy is reduced with 43% compared to the MUL version.

However, this conversion introduces an error into the FFT, which can only be allowed if the application does still perform within spec. In this case, the FFT was part of an $IEEE$ 802.11 n 2 antennas MIMO receiver. This system supports a set of different communications modes, as shown by Figure 10.22, ranging from a SISO BPSK (Single In Single Out, Binary Phase Shift Keying) to a MIMO QAM 64 mode (increasing complexity).

A detailed Matlab accuracy simulation has shown that for all but the two most complex modes, the strength reduction with a single SSA operation is providing sufficient accuracy. The $MIMO\ QAM$ 16 and $MIMO\ QAM$ 64 modes with 1 SSA show an increased Bit Error Rate (BER) for high SNR (shown at the right side of Figure 10.22) with respect to the strength reduced versions with 2 SSA operations per MUL. This indicates that the introduced

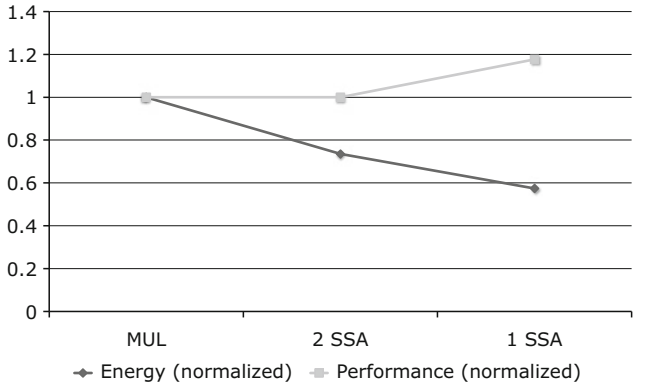


Figure 10.21: Impact of MUL-conversion with a maximum of 2 inserted SSA and 1 inserted SSA operation respectively, on energy and performance of the FFT kernel, normalized to the MUL version

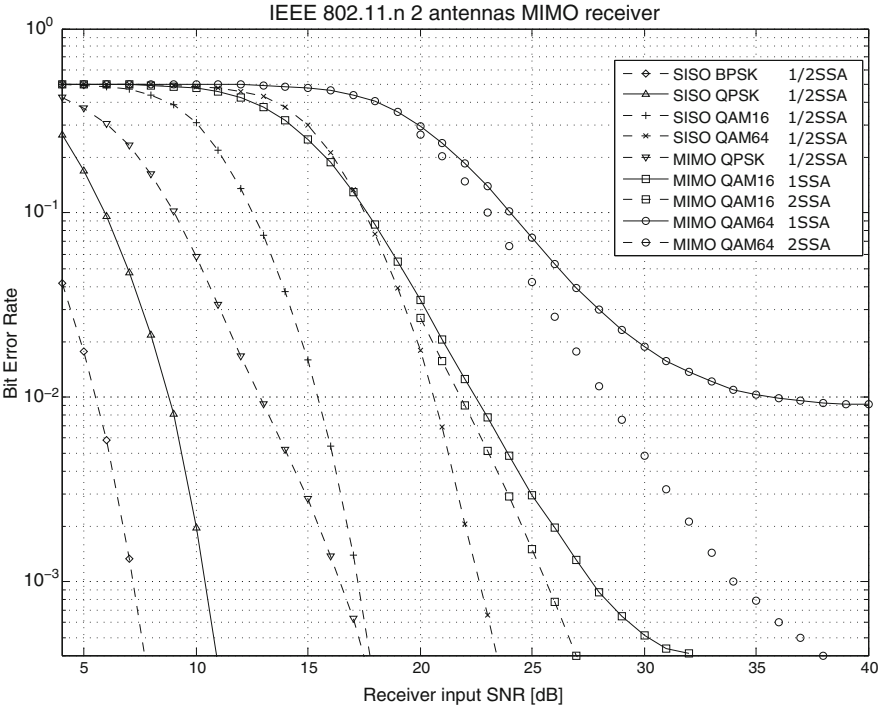


Figure 10.22: SSA accuracy for FFT in SISO/MIMO implementation for different modes, using 1 or 2 SSAs per MUL

error in those modes leads to a degradation in the application performance. When using these modes, more operations have to be spent, while all other modes can save energy by using a cheaper conversion.

The strength reduction for the FFT kernel of this section has shown that significant trade-off opportunities exist, when exploiting application requirements during the strength reduction. Additional opportunities exist when this trade-off is extended to the architecture exploration during processor design, as is shown in the following section.

10.4.4 DWT kernel, part of architecture exploration

In this section, the conversion of constant multiplications is explored as part of the processor design exploration and together with the application accuracy requirements. This step can be integrated into the ASIP architecture exploration when deciding how many MUL-capable slots are required with respect to the requirements of the target application domain.

In this case, the target application is a Discrete Wavelet Transform (DWT) kernel, as is used in the JPEG 2000 [JPEG] image encoding or in several biomedical algorithms[Sas95] used for, e.g. compression of Electrocardiogram (ECG) signals. The specific DWT that is used here [DWT] is a Discrete bi-orthogonal CDF 9/7 wavelet forward transform (lifting implementation) that consists of 8 subsequent loops and contains a set of filtering steps with constant multiplications.

The original floating point benchmark has been converted to a 16-bit fixed point version, which resulted in an error with respect tot the floating point description. In this specific case, the introduced peak error for the multiplier implementation was equal to 7.62%, as indicated by Figure 10.23. The specific constants of the DWT required in this case 4 SSA operations per MUL for an exact conversion (equal to the MUL version) using the conversion strategy of Section 10.3.7. By reducing the number of inserted SSAs, an additional error has been introduced, as indicated by Figure 10.23. The version with 3SSAs, even though some constants are approximated, does not lead to an increased peak error. When only 2 SSAs are used per MUL operation, the peak error slightly increases with about 2%. When every MUL is replaced by a single SSA, the error rises significantly with an additional 11%. Depending on the application requirements this may or may not be tolerated.

In this section, the link between the different implementation options for the multiplier accuracy and the architecture design is explored. The rationale for this trade-off could be the impact of the number of multipliers on the area (and therefore the cost) of the processor core, but also indirectly the impact of the datapath area on the length of the processor interconnect, which e.g. can have an important effect on the energy consumption in the

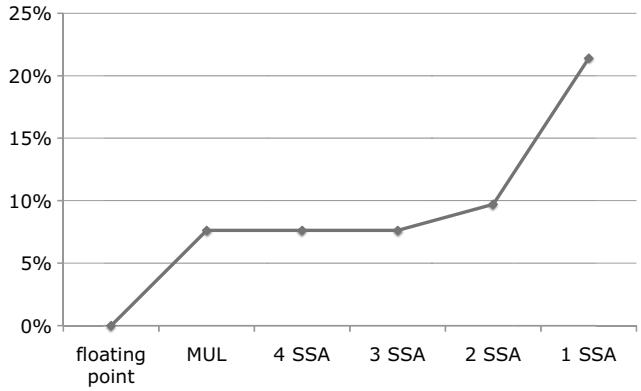


Figure 10.23: Introduced error of different MUL implementations, compared to floating point version

interconnect of CGRA processors (See Chapter 4 of [Lam09]). In many cases the number of multipliers in the architecture is decided based on the algorithmic requirements, often explored at Matlab level, e.g. the ADRES instance optimized for wireless communication architectures of [Bou08] consists of a 4×4 CGRA in which every PE includes a multiplier. As the area of a 16-bit multiplier is 15 times larger than the area of a 16-bit adder,[4] it makes sense to investigate the strength reduction in the context of architecture design. Many wireless algorithms heavily depend on linear transformations like FFT's and many multiplications can be strength-reduced (as shown for the FFT above) to reduce the number of PEs that include a multiplier. A second example of this trade-off can be found in the biomedical context, where extreme energy efficiency is targeted for very small cores. The signal processing includes a large amount of filtering and linear transformations, in which context a large amount of multiplications are present.

In this experiment, different versions of a (manually) heavily optimized DWT kernel have been mapped on different VLIW processor architectures, assuming that a reduction in the number of multiplier slots could lead to an area cost reduction, which in turn could be exploited by adding more non-multiplier slots. As the indirect effect on the processor interconnect energy consumption is difficult to estimate for a VLIW processor, the experiment will focus on performance improvements for different processor variants. The presented results have been achieved for a loop unrolling factor of 4.

Figure 10.24 shows the results of this experiment, assuming an initial VLIW processor with three slots, of which two include a multiplier (e.g. based on high level application requirements). This architecture is compared with other architecture variants in which one multiplier has been removed and ad-

[4] Based on in-house a standard-cell design in TSMC 40 nm.

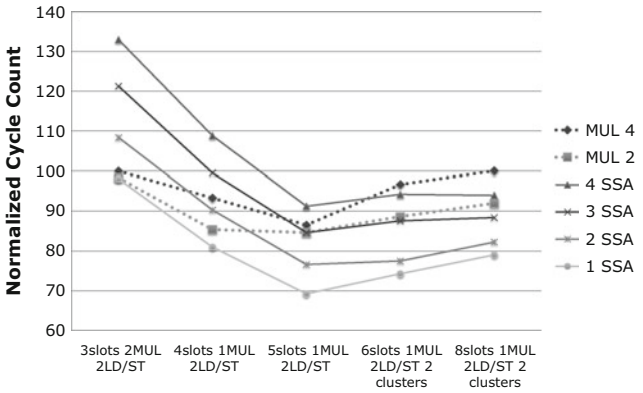


Figure 10.24: Performance comparison for DWT MUL conversion exploration (the six- and eight-slot architectures use a clustered register file with two clusters)

ditional non-multiplier slots have been added, in case a 4-, 5-, 6- and 8-slot variant. This corresponds to 1 to 5 additional slots, based on the fact that an adder is 15 times smaller than the removed multiplier, but an additional overhead of an ALU slot (all added slots support only ALU operations) and other control overhead needs to be taken into account.

The size of the register file has been kept constant, but as the number of port needs to increase, the 6- and 8-slot variants have been clustered (the area increase of the RF with an increasing number of ports also has to be taken into account). Other architectural parameters (e.g. slot functionality and number of load/store units of the processor) have been kept constant.

In this case, the performance results of Figure 10.24 (normalized to the initial architecture) show some interesting trade-off opportunities, linked to the application accuracy requirements of Figure 10.23. When comparing the DWT version using a multiplier with a latency of four cycles (MUL 4), trading in the multiplier for one additional slot leads to significant performance improvements for the versions with 2 and 1 SSA operation per removed MUL (about 5% and 15% respectively). For a multiplier latency of 2 cycles (MUL 2), a performance gain can be achieved for the version with 1 SSA for the first slot, but when more slots are added, e.g. five slots, the 3 SSA version achieves the same performance and the 2 SSA and 1 SSA version achieve improvements of about 10% and 20% respectively. It is interesting to note that moving to the clustered architectures with six and eight slots degrades the performance due to the inter-cluster communication overhead, but the negative effect is larger for the MUL-code as the single multiplier in the first cluster becomes more of a bottleneck, while the strength-reduced code can still distribute the ALU operations over multiple slots.

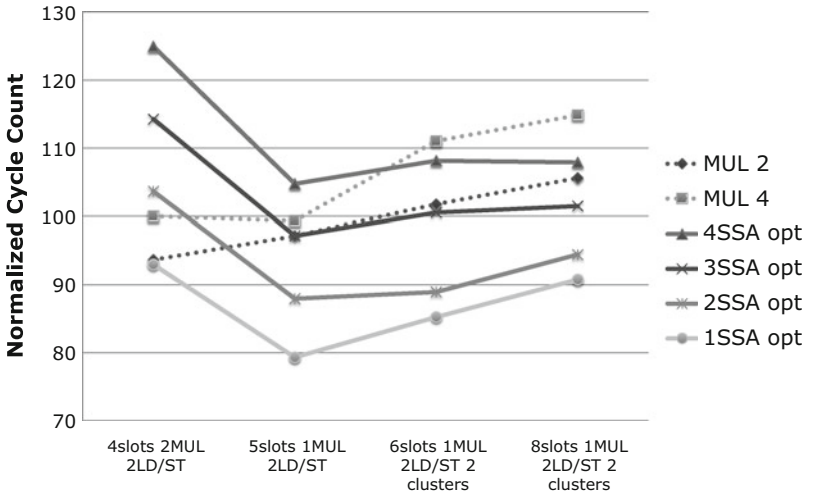


Figure 10.25: Performance comparison for DWT MUL conversion exploration (the 6- and 8-slot architectures use a clustered register file with two clusters)

As the multiplier code also shows a limited benefit from the addition of one or two ALU slots instead of an additional multiplier, it is clear that the DWT kernel is not fully multiplier bottlenecked. However, the strength reduction still achieves nice performance improvements for this code.

Figure 10.25 shows the same experiment, but for a different initial architecture with four slots and two multipliers. In this case, removing one multiplier and adding a single slot already results in equal or better performance for the 3 SSA, 2 SSA and 1 SSA versions. The performance improvement is also significant, e.g. over 10% for the 2 SSA version on a 6-slot architecture with respect to the multiplier version with latency 2 (MUL 2).

It should be noted here that the quoted performance improvements are significant, as the initial code has been heavily manually optimized before the strength reduction was performed. The improvements, starting from a very good baseline, lead to extremely efficient code, as is shown in the example schedule of Figure 10.26 in which only four empty cycles are available in the software pipelined schedule for one of the loops of the DWT kernel.

10.4.5 Online biotechnology monitoring application

This application has been worked out in detail in Chapter 11, including the constant multiplier transformation into shifts and adds.

Basic block 11

RecMII no spatial= 7 RecMII= 7 ResMII= 10 numnodes= 46 ESC= 0 II= 10

	0_1	0_2	0_3	0_4	0_5
1	ADD_W	ADD_W	ADD_W	ADD_W	ADD_W
2	ADD_W	L_W_C1_C1	ADD_W	L_W_C1_C1	ADD_W
3	ADD_W	L_W_C1_C1	ADD_W	L_W_C1_C1	ADD_W
4	SSA	SSA	SSA	SSA	SSA
5	SSA	SSA	SSA	SSA	ADD_W
6	SSA	ADD_W	SSA	ADD_W	SSA
7	ADD_W	ADD_W	ADD_W	ADD_W	ADD_W
8	ADD_W	SHRA_W	SHRA_W	SHRA_W	ADD_W
9	SHRA_W	S_W_C1		S_W_C1	
10	BRF_B_B_F	S_W_C1		S_W_C1	

Figure 10.26: Example of heavily optimized DWT schedule after MUL conversion

10.4.6 Potential improvements of the strength reduction

The presented experiments have demonstrated the constant strength reduction for standard architectures. However, when other architectural extensions are present, potential improvements over the demonstrated approach can still be achieved. This section briefly discusses some of these potential improvements.

10.4.6.1 Loop Buffer with Local Controller

The converted multiplications are in most cases part of a loop in which the constant varies over the loop. This has some consequences on how the conversion can be done, as, e.g. the number of inserted operations after strength reduction can not be varied from one constant to the next. Therefore, the experiments presented in the previous section have fixed this number, independent of the constants. This means that if the number of inserted operations is fixed to, e.g. 3, constants that only require 1 operation for an exact conversion will still cost 3 operations.

When the architecture contains loop buffers with local controller [Jay02a], the compiler can generate different versions, based on how many operations are required and at run-time the execution can jump to the correct entry, thus reducing the execution cost and improving the performance, without accuracy loss. The importance of the use of distributed loop buffers has been strongly shown in the biotechnology monitoring experiment of [Kri09]. In particular, due to this extension, the energy consumption of the instruction memory hierarchy contribution in the Software SIMD ASIP mapping has been reduced to a negligible overhead.

10.4.6.2 Link between SSA, CSD and performance

The presented conversion experiments mostly target performance improvements and therefore make use of a conversion strategy that tries to find the

cheapest conversion with minimal depth of the dependence chain. The instruction set has been fixed to a rather standard instruction set, extended with the SA or SSA operations. This results in a conversion that in most cases favors a CSD based conversion, combined with the binary method. However, depending on the target clock frequency and the technology node, more challenging special FUs can be designed, of which the Generic Shift-Add-Shift or GSAS is one example. In that case, depending on if the more complex operations will be possible with a single-cycle latency and if they can be pipelined, this will result in very interesting conversion optimization problems.

10.4.6.3 Multiple precision MUL operations

All presented strength reduction experiments have been using a full conversion, meaning that all constant multiplications have been converted. However, in some cases (that require a high accuracy for a very large constant) the conversion can be expensive and a multiplication can still be the cheapest option. To cover those cases and the multiplications that can not be converted into constant multiplications, most processors will still require a multiplier to be present in one of the slots. However, as the ratio of non-converted multiplications to other operations will be low, it is possible to replace this expensive multiplier by a multiple precision version: e.g. a 4-bit MUL instead of a 16-bit MUL, with the option of using the 4-bit version to perform 16-bit multiplications with increased latency. In this case, 16 operations on the 4-bit multiplier are required to generate one 16-bit result. This will lead to additional gains in energy efficiency due to interconnect length reduction and especially in area, with limited performance degradation. The resulting architecture is still generic, as it still supports variable multiplications.

10.5 Comparison to related work

Strength reduction techniques for optimizing compilers for high performance systems have been studied extensively in the past. [Coo01] gives a good overview of related work on operator strength reduction. Most techniques are restricted to constant multiplication with iterator or induction variables of loops. One class, based on [All81] and [Coo01], targets optimizing constants and loop-invariants for a single loop exploiting control flow analysis information. Another class, e.g. [Dha79], is based on data-flow methods. These techniques are complementary, as this chapter targets constant multiplications to data variables other than iterator or induction variables. In the results presented in this chapter, the state-of-the-art optimizations have been enabled in our compiler front-end [Rag08b] and the shown improvements are with respect to this state of the art.

Other techniques specifically target the strength reduction of constant multiplications. [Har91] and [Par01] focus on digital filter design, while [Ber86] and [Par01] have proposed conversion techniques for general purpose processors. A large amount of related work exists in the context of Canonical Signed Digit (CSD) conversion methods [Hew00], including Binary Coded CSD (BCSD) [Has96] and Minimal Signed Digit (MSD) [Par01] representations. Interesting conversion strategies include the localization of $optimally\ convertible\ blocks$, to reduce the search space, but the conversion is completely decoupled from the underlying instruction set and the application requirements. They present a limited overview of the different conversion options (e.g. binary and sequential), but only consider a search cut-off heuristic with a very limited instruction sequence cost. This basic algorithm has been generalized in [Bri94, Lef01] and is currently implemented in GCC, a state-of-the-art compiler. However, this implementation considers only constants or its factors which can be broken down into the canonical form $(2^m \pm 1)$.

[Wu95] also proposes a search over the conversion space steered by heuristics, using a branch and bound approach that favors one specific instruction set extension, called LEA, as much as possible. The conversion targets only performance, is implemented in an x86 compiler, but it does not consider coding options like CSD and therefore misses important optimization potential. This approach also misses a systematic description of the search space and therefore does not cover options like CSE and additive factoring.

Techniques like [Kar07] explores the complexity reduction in FIR filters from the point of view of removing computational redundancy in the synthesis phase of multiplier less filters. They synthesize "modified coefficients" so that a proper trade-off between computation/communication complexity and filter quality can be made. This corresponds to propagating the applications requirements to the conversion. However, the rest of the conversion space is not explored and the specific context of a processor based implementation is missing in this work.

To the best of our knowledge, no single technique exists that covers the complete relevant search space in a systematic way (as described in Section 10.3), while taking both the underlying instruction set as well as the application requirements into account, which enables more complex conversions.

10.6 Conclusions and key messages of chapter

Multiplications are expensive operations, in terms of area, energy consumption and cycles/performance. Strength reduction is a well-known technique to convert a subset of all multiplications into less costly operations. However, the state-of-the-art strength reduction techniques only convert the most

simple cases and can still be improved by taking into account other design considerations. Some examples include firstly, the removal of all multiplications, even at a relatively high cost, to enable other optimizations, like Software SIMD. Secondly, by taking into account and potentially extending the processor instruction set, the cost of the converted operations can be reduced. Thirdly, by propagating the application requirements to the conversion, the accuracy of the multiplication can be traded-off for the implementation cost.

The work that is presented in this chapter contains several contributions and it is different from the related work in the five following aspects.

Firstly, it presents a complete and systematic classification for the conversion of constant multiplications to a sequence of less complex operations. The space is represented as a set of trees, in which a full conversion method is fixed by a combination of decisions in each of the trees. Secondly, the strength reduction is explicitly linked to the underlying hardware, as this heavily influences the final cost of a conversion. Based on the processor instruction set, decisions in other trees can be made or extra transformations can be used, in order to reduce the total conversion cost. During architecture exploration, special instructions can be considered to reduce this cost even further. Thirdly, the accuracy requirements at the application level are taken into account, in order to reduce the cost of the conversion. Traditionally, only I/O true transformations are considered during the mapping. However, by verifying the impact of a reduced accuracy in the operator implementation at the application level, in many cases the introduced error can be tolerated and larger cost reductions can be achieved. Fourthly, the conversion of constant multiplications is considered in the context of being an enabler for other optimizations that require multiplication-free code (e.g. Software SIMD). In this case, the cost of the conversion can be tolerated to be high, if the enabled optimization will result in an even larger cost reduction. Finally, a search strategy over the described conversion space is proposed. A cost-aware search over the described conversion enables a context-specific conversion, based on the relative importance of multiple quality criteria (e.g. improve performance vs. reduce energy, target a very parallel processor vs. a less parallel processor). This approach is demonstrated using a conversion technique based on a simple heuristic to improve performance. By changing this cost function, the conversion can be applied in different contexts and is therefore not limited to the specific examples or architecture shown in this chapter.

CHAPTER 11

Bioimaging ASIP benchmark study

Abstract

This chapter describes the application of the main techniques proposed in this book to a realistic application benchmark, namely a bioimaging detection and tracking algorithm for on-line animal monitoring. Most of the components and contributions presented in this book have been applied and illustrated in this realistic demonstrator. In particular we exploit the distributed loop buffer organisation, the very wide register with a wide interface to the SRAM scratchpad, and the SoftSIMD concept in the data-path including the constant multiplication strength reduction. All these are embedded in an instance of the FEENECS *architecture template of Chapter 3.*

First the application is discussed in Section 11.1, including the fixed-point word-length quantisation of the variables. Section 11.2 discusses our effective realisation of the many constant multiplications in the application code. Section 11.3 describes the different architecture options that are energy and performance optimized for the scalar ASIP template. Similarly, Section 11.4 describes the options for the data-parallel ASIP template, with emphasis on

F. Catthoor et al., *Ultra-Low Energy Domain-Specific Instruction-Set Processors*,
DOI 10.1007/978-90-481-9528-2_11,

the data-path. Section 11.5 continues with the background and foreground memory organisation for that data-parallel ASIP. Section 11.6 concludes with the overall energy results and the discussion of the global ASIP exploration.

11.1 Bioimaging application and quantisation

Biotechnology is defined as the technology that deals with the living cells and its purpose is the improvement of the human living and human plus animal welfare. Today, biotechnology has a significant impact on the world economy and the world society, since it is applied in many daily activities. Biotechnology applications are divided in four major sectors, namely: health care, agriculture, industrial products and environment.

A part of agriculture biotechnology applications finds use in livestock farming and especially in animal observation. The latter has proven to be very useful in the improvement of their productivity, physiology and health. A common way of observing a living organism is usually done by audio-visual ways performed by a human who is present on the scene. This method is however subjective, expensive, error prone and time consuming. Relying on the fact that the correctness of the results crucially depends on the capabilities of the observer and on the way he interprets the behavior of the animal, the human factor is directly or indirectly causing both subjective observations and the erroneous conclusions that may occur. Moreover, the person should be present through the entire time period of the observation, a fact that makes this method expensive, both in time and in cost. Instead of performing the animal observation by a human, on-line monitoring systems are proposed. In the benchmark selected in this book chapter, we are using specifically the bioimaging approach proposed by the BIORES group at the K.U. Leuven [Ler06].

On-line monitoring systems are capable of collecting information about the environment that they are observing. Analyzing these data, they use the results to specify many aspects of the monitored object, such as its position, its movement or even its behavior. They consist of low-cost intelligent sensors that are combined with image analysis techniques to provide an automated objective contact-less monitoring method for the behavior of the living organisms. It is based on a computer vision application categorizing the behaviour of animal responses. In the instantiation used here, it has been tuned to monitor individual laying hens. A video camera is used as an input device to the system that provides images taken from the hen. It should be mentioned that the current case study is performed on a video sequence of 3 s, with 25 frames/s. This bioimaging approach is separated in two algorithmic modules. The first module detects the monitored object (detection algorithm) and the second one tracks it (tracking algorithm). During the detection

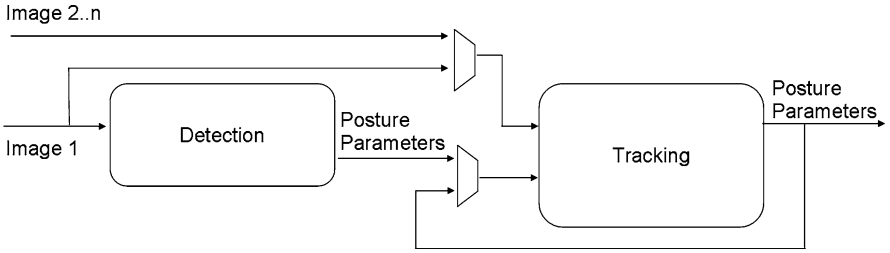


Figure 11.1: Bioimaging application flow

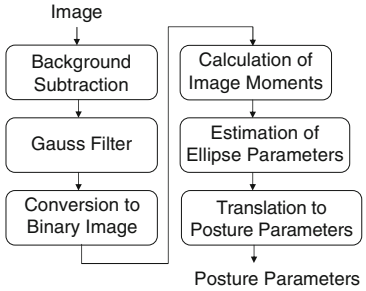


Figure 11.2: Detection algorithm flow

algorithm, the monitored object is being detected using image processing techniques on a frame captured by a video camera. Then, the tracking algorithm is executed in order to locate, each time, the position and the characteristics of the monitored object that have changed. These changes over its characteristics are categorized and finally translated to the behavior of the monitored animal. In Figure 11.1, a brief description of the flow of the bioimaging application is provided.

The detection of the monitored object is performed on the first input frame by an image processing algorithm. The algorithm determines the position, the orientation, the body length and width of the monitored object through a set of parameters that are based on an ellipse shape model and are called ellipse parameters. After their calculation they are translated to posture parameters, which are the final output of the detection algorithm. The flow of the detection algorithm of the bioimaging application is illustrated in Figure 11.2.

A typical input image is shown in Figure 11.3. In the first step of the detection algorithm, the first input grayscale image is subtracted pixel by pixel from the background image and then a Gaussian filter is applied to the result of the image subtraction in order to reduce the existing noise. The application of the Gaussian filter is performed by the multiplication of the Gaussian coefficients by the appropriate neighbor pixel, horizontal, vertical and diagonal.

Figure 11.3: Ellipse fitted to the body of the hen

Then the sum of these multiplications replaces the value of the central pixel. The result of the application of the Gauss filter in the whole image is its blurring in order to create a less noisy one. It should be mentioned that the calculation of the ellipse parameters of the monitored object, and hence the calculation of its posture parameters, demands the image to be in a binary form. This binary conversion necessitates the determination of a threshold value, that depends on the quality of the images, in order to distinguish the "hen" pixels from the "background" ones. The conversion consists of reading the whole image pixel by pixel. In case its value is less than the estimated threshold value, then it is converted to a black pixel, which means a background pixel with logic value equal to 0. In case its value is more than the threshold value, then it is converted to a white pixel, which means a hen pixel with logic value 1. Then, the ellipse parameters are evaluated using the binary image. Their calculation is based on the indication of a shape that can approximate the two-dimensional object of the binary image. Studying the images that derive from the imaging processing steps, a simple ellipse shape is used for the approximation of the monitored object's body. As illustrated in Figure 11.3, this ellipse is calculated in order to best fit into the shape of the monitored object.

The ellipse is a simple shape that is used for estimation of the hen's contour and it is modified through the variance of five parameters, which are called ellipse parameters. These ellipse parameters, which are depicted in Figure 11.4, consist of the coordinates of the ellipse centre of the X_c and Y_c, the orientation angle θ, the major axis l and the minor axis w. The position of the ellipse is given by the position of the center; while its orientation corresponds to the angle θ, which is the angle between the major axis l of the ellipse and the x axis. The size of the ellipse is determined by both the major axis and the minor axis, l and w respectively.

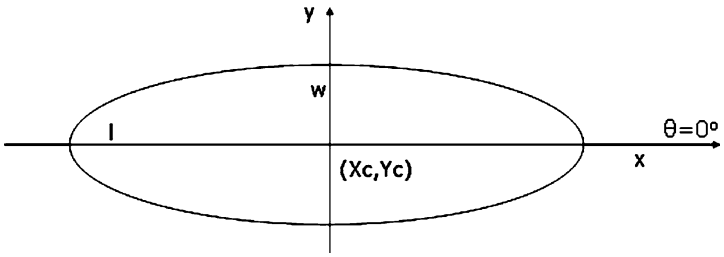


Figure 11.4: Ellipse model

Their calculation is based on the calculation of the general two-dimensional order moments p and q of the binary image [Roc02].Then, the ellipse parameters are translated to posture parameters, which consist of the coordinates of the centre X_c and Y_c, the parameters r_x and r_y, and the parameter p. The coordinates of the ellipse center, X_c and Y_c, indicate the position where the ellipse is placed in the image. The parameters r_x and r_y represent the scale and the rotation factor of the ellipse, whereas the parameter p represents the factor that estimates the transformation of the ellipse in order to fit into the monitored object. The five posture parameters are evaluated based on Eqs. 11.1 and 11.2.

$$X_c = \frac{m_{10}}{m_{00}}; Y_c = \frac{m_{01}}{m_{00}} \tag{11.1}$$

$$r_x = (l+w)\cos\theta; r_y = (l+w)\sin\theta; p = \frac{l-w}{l+w}; \tag{11.2}$$

The posture parameters are the final output of the detection algorithm. These are inserted as input arguments to the tracking algorithm, where they are dynamically updated through a set of steps. The tracking algorithm is applied a number of iterations on the same input frame, i.e. ten iterations per frame (10 iterations/frame), in order to optimize the results. The updated posture parameters are fed back again to the tracking algorithm which now is applied to the next successive frame. This continuous update of the posture parameters consist the main idea of the way the algorithm tracks the monitored object.

Due to the non-demanding image processing analysis of the tracking algorithm, it has faster execution time and significantly lower energy consumption than the detection algorithm. Actually, the number of operations per frame of the tracking algorithmis a factor 50 lower than the one of the detection. Accordingly, only the tracking algorithm is applied for many successive input frames in order to reduce the image processing time and the energy consumption. The number of frames where only the tracking algorithm is applied and no detection, should be low enough so that it does not degrade the performance of the overall bioimaging application and hence where the monitoring results are considered acceptable. As already mentioned, the video

sequence lasts 3 s with 25 frames/s. Within the 3 s, in our instantiation the detection algorithm is applied once in the first frame and then the tracking algorithm is applied 10 times for each of the next 75 frames of the video sequence, so the applied ratio between the detection algorithm and the tracking algorithm is 1/750. The bioimaging application, is used in on-line monitoring systems, which means that the performance of the application must meet some hard real-time performance requirements. We are assuming here the image contains only a single chicken that should be monitored continuously (no time multiplexing between chickens). Considering a realistic processor clock frequency of 200 MHz, the available time between two consecutive frames is equal to 0.04 sec and the available cycles for performing both the detection and the tracking algorithm amount to 0.04 s $\times$ 200 Mhz $= 8{,}000{,}000$ cycles $= 8$ Mega Cycles.

This bioimaging application aims at a low energy and low cost implementation and will now be targeted to an ASIP platform, that is the core of an embedded system. Such processors do not use floating point Function Units (FUs) due their high cost and high energy consumption. Consequently, the bioimaging application must be converted from floating point to fixed point representation before it is mapped to the potential ASIP platforms during architecture exploration. In order to apply this conversion, it is obligatory to determine the appropriate word-length of the application variables. This is performed by quantization analysis which determines the smallest acceptable word length for each signal, i.e. while maintaining the output results of the bioimaging application to be acceptable. The quantization analysis is based on calculating the required range (dynamic range analysis) and adjusting the accuracy (precision analysis) of each one of the bioimaging application signals. The estimated word-lengths are used for the transformation to the fixed-point representation.

For this purpose, we have used a systematic and effective approach that is described in [Nov08, Nov09]. A brief summary of its application will now be provided here but the details have been left out as this step is not the focus of this book. The determination of the dynamic range of the signals is based on the annotation of the maximum value, positive or negative, that is assigned to these signals during the execution of the bioimaging application for a representative input stream. It gives the appropriate bits for the representation of the integer part of each signal. Regarding the required bits for representing its decimal part, precision analysis is performed. It consists of a number of steps that modify the number of the decimal bits in such a way as to determine iteratively the exact word-length that will be used by each bioimaging application signal.

The calculation of a baseline solution for the word length of each signal of the bioimaging application is done by an exploration of the solution space for finding the near optimal solution. A set of bounded CPU-time heuristic

methods have been used whose priority function is the reduction of the word length of the most important signals. A measure that reflects the importance of each signal is the total cost of the operations in which it takes part. The cost of each signal is based not only on the amount and the kind of arithmetic operations, but also on the memory allocation and on the amount of the memory operations. Different weights are used to indicate the most important signals in the application. These heuristics reduce the computational problem of exhaustive exploration in order to obtain an near optimal solution, which is depicted in Figure 11.5. The baseline solution is placed between the Real Lower Bound solution (RLB) and the Upper Upper Bound (UUB) solution. The RLB solution stands for the lower bound of the word length that each signal of the bioimaging application can have and the results are acceptable, while the rest of the signals are having their UUB. The UUB stands for the "fully safe" higher bound of word length that each signal can have and that leads to a potential reduction in the word length of another signal. In the rest of this chapter, the calculated word lengths of the baseline solution are used in order to represent the bioimaging application in fixed point arithmetic. More information is available in [Kri09].

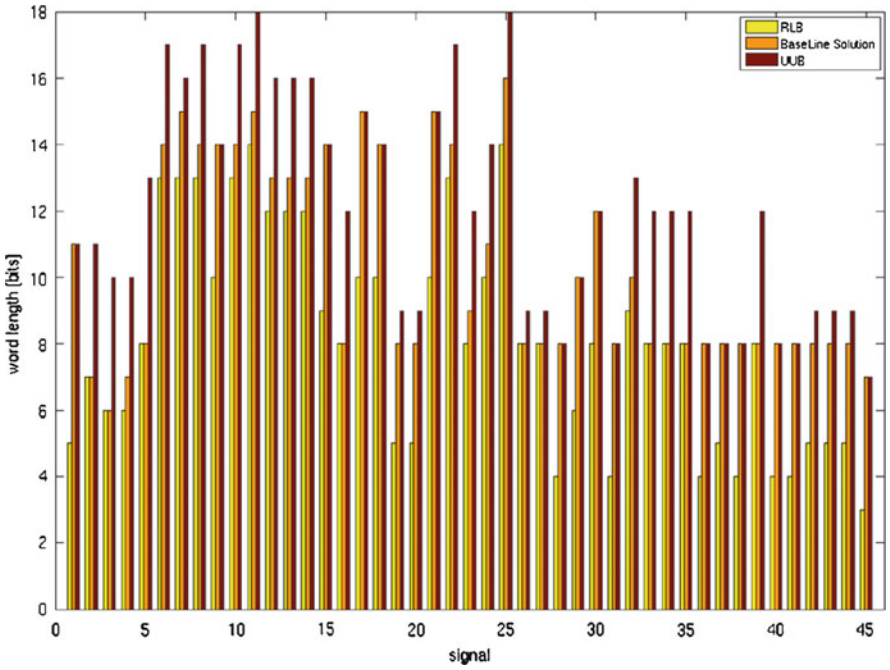


Figure 11.5: Total word length (y axis) for each signal (x axis) of the detection algorithm of the bioimaging application. The bars show the lower bound RLB (green), baseline solution (red) and the upper bound UUB (black)

11.2 Effective constant multiplication realisation with shift and adds

The multiplication is a functional operation that is widely used by the programmers and in the processor target, it usually is implemented by a multiplier Function Unit (FU). Although the multiplier FU has high performance, it increases the cost and especially the power consumption of the processor, due to its domination in the data path area and energy consumption. This is a very important fact for the on-line bioimaging application which aims at a low energy and low cost implementation for use in chicken farms. This means we should avoid as much as feasible the use of the multiplier FUs in the final ASIP platform. Hence, in this book we have studied a more efficient implementation based on the replacement of constant multiplications by sequences of shift and addition operations (see Chapter 10) that are implemented on specific FUs.

First of all, the efficiency of this replacement of operations is examined in order to decide where this conversion is worthwhile to be applied. Theconversion efficiency is determined by profiling the code of the bioimaging application. The more constant multiplications exist in the bioimaging application, the more gain will be obtained by this conversion. The total number of the arithmetic operations and the multiplications of the detection algorithm are illustrated in Figure 11.6.

The Arithmetic and Logic operations (ALU), which do not include multiplication operations and divisions, are dominant in both the detection and the tracking algorithm. It is stressed that the profiling of the tracking part is done on one iteration of the algorithm. Depending on the iterations performed by the algorithm, the above results are multiplied by the appropriate

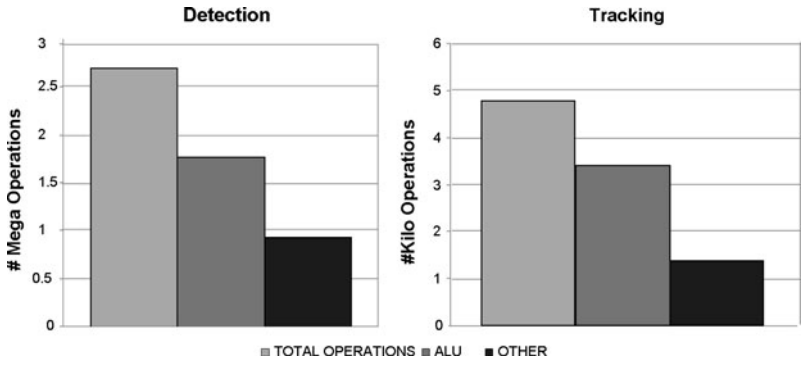


Figure 11.6: Total number of arithmetic operations of the detection and the tracking algorithm

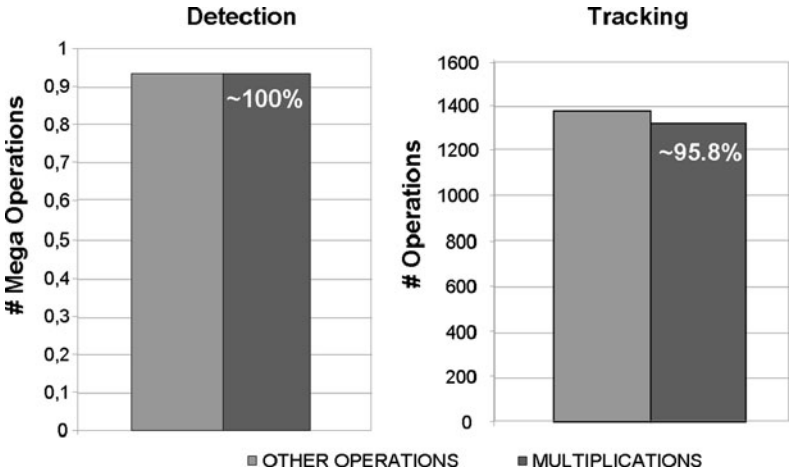


Figure 11.7: The multiplication operations for the detection algorithm represent almost 100% of the non-ALU operations and for the tracking algorithm about 96%

factor. According to the application's specifications (video sequence of 3 s, 25 frame/s, 10 iteration/frame), the appropriate factor is equal to 75×10. A more detailed profiling of the remaining operations, i.e. other operations, is needed in order to determine what kind of operations they are. The majority of the other operations (see Figure 11.7) are multiplication operations for both the detection and the tracking algorithm of the bioimaging application. In addition, in Figure 11.8 the number of multiplications that corresponds to constant multiplications for both algorithms is illustrated. Observing these graphs, the constant multiplications are the main operations of the remaining operations (other op), e.g. the total operations excluding the ALU operations that by default do not include the multiplication or division operations.

The conversion of a constant multiplication to a sequence of shift and addition operations can be performed through different approaches, such as the parallel, the sequential and the hybrids. The sequential approach is typically implemented by using left-shift operations and hybrids, as described in Chapter 10. But alternatively it can also be implemented by using right-shift operations. Especially in the SoftSIMD context, the right-shift sequential approach is more advantageous than the left-shift sequential one, because it reduces the needed data-path width. During the right-shift sequential approach the digits that are calculated first are the LSB bits of the output. The right shift operation pushes the LSB bits of the partial results out of the data path and they are lost. Moreover, the right-shift sequential approach preserves the necessary precision, while it uses a data path with width equal to the smallest one needed, i.e. the output size. In case of left-shift operations, it is necessary to increase the data path width, since the left operations push

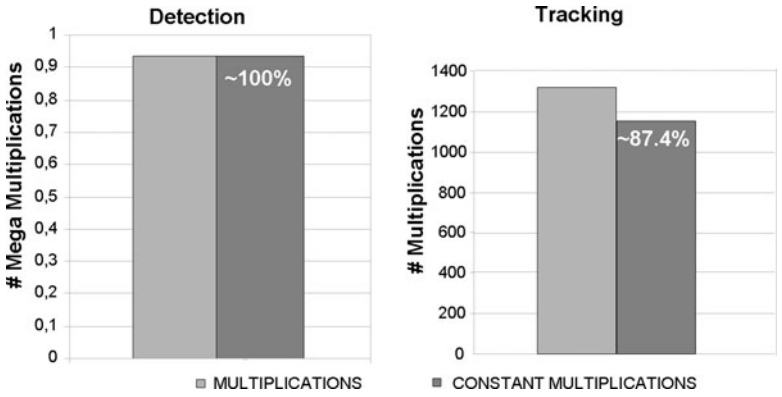


Figure 11.8: The constant multiplications for the detection algorithm represent almost 100% of the multiplications and for the tracking algorithm they amount to 87.4% of the multiplications

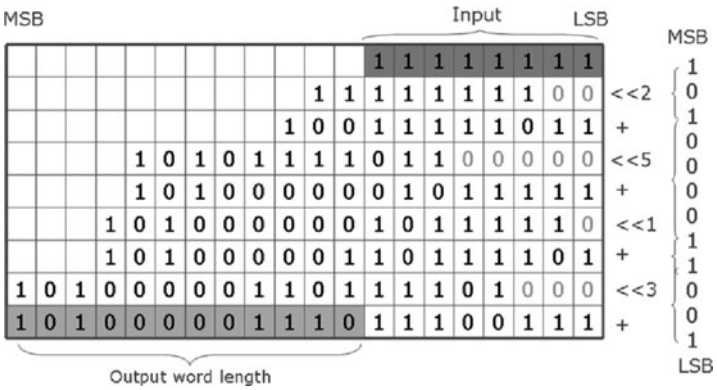


Figure 11.9: Multiplication of $255_{dec} \times 2585_{dec}$ using the left-shift approach

the MSB bits of the partial results out of the data path and the precision of the output is not guaranteed. The correct implementation of the constant multiplication using left-shift operations needs a wider data path than the implementation with right-shift operations. After the calculation of the final result, the output is shifted back to the precision needed. A more detailed example $(255_{dec} \times 2585_{dec})$ that uses the left-shift sequential approach is shown in Figure 11.9.

In this example, the input of the multiplication, e.g. the multiplicand, has a word-length equal to 8 bits while the output has a word-length equal to 12 bits. In order to have a correct output, it is required that the data-path width is equal to 20 bits, although the needed word-length of the output is

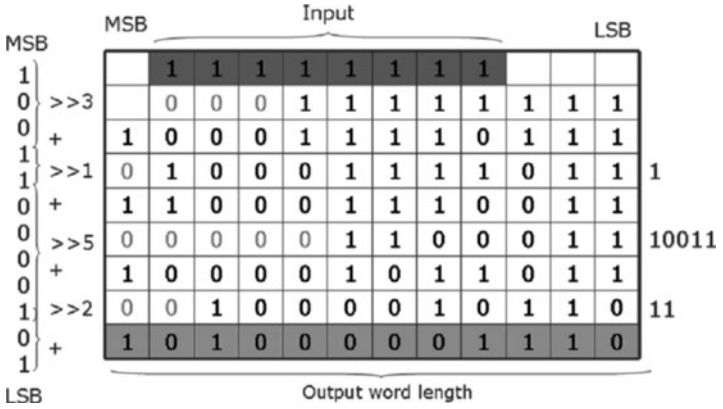


Figure 11.10: Multiplication of $255_{dec} \times 2585_{dec}$ using the right-shift approach

only 12 bits. The use of left-shift operations moves the MSB bits of the partial product out of the needed output word length and as a consequence, the data path is growing. After the calculation of the output it is shifted back to the appropriate word length. In the case of the right-shift sequential approach, shown in the example of Figure 11.10, the input is aligned to the MSB $-$ 1 bit of the output. The use of right-shift operations moves the LSB bits of the partial products out of the data path and as a consequence of this, the result is correctly calculated, while the data-path width is only 12 bits, i.e. the word-length of the final output.

Tradeoffs exist between these different approaches regarding the performance, the energy consumption and the area. It is stressed that the analysis of the constants of the bioimaging application that are used in multiplications, is necessary in order to determine which approach and how many FUs are necessary for the implementation of the sequence of shift and addition operations. Hence, constant analysis is carried out both for the parallel approach, which has better performance but increased hardware demands and the sequential approach with right shifts, due to its advantages in area and energy consumption. The constant analysis includes the indication of all the different constants of the application that take part to constant multiplications and they are either positive or negative values. Only the case with the positive constants is further analyzed here, since the negative ones can be transformed. The multiplication by a negative constant is replaced by the multiplication by the absolute value of the constant and the sign is then propagated to the next addition/subtraction input and absorbed there. The positive constants are determined by their binary value, which provides the needed shift factors and the necessary additions according to its non-zero digits (see Chapter 10). Consequently, the cost of implementing the sequence of shift and addition operations is given by the number of non-zero digits of the corresponding constant.

For performing the constant analysis, the utilization of the constant with the worst cost is a critical factor for deciding the use of the worst case or the average case [Kri09]. The constant with the worst cost used in a multiplication of the bioimaging application is 32767_{dec} and it is part of the tracking module. Considering the fact that the detection algorithm is applied once over 75×10 applications of the tracking algorithm, according to the application's specifications (video sequence of 3 s, 25 frame/s, 10 iteration/frame), the utilization of the worst case constant is not negligible and, consequently, drives the constant analysis. This worst cost constant is equal to 15 non-zero digits, which means that 14 shift operations and 14 addition operations are needed in order to convert this multiplication to a sequence of shift and addition operations. The high requirements in shift operations and in addition operations lead to the selection of another coding, namely the Canonic Signed Digit (CSD) coding (see Chapter 10), which represents the same value but with less non-zero digits. Hence, the worst cost using the CSD coding has changed to 23170_{dec} and it is equal to six non-zero digits, which means that five shift operations and five addition operations are needed in order to convert the multiplication by the worst case constant to a sequence of shift and addition operations.

Moreover, the worst case constant analysis with different approaches for the implementation of the sequences of shift and addition operations is performed in order to determine the most promising sequence. First of all, the multiplication by the worst cost constant is converted to a sequence of shift and addition operations considering a parallel approach, as illustrated in Eq. 11.3. Three shift-shift-add (SSA) FUs are necessary for achieving the highest performance, but this incurs a large cost in hardware requirements. The mapping of the worst case sequence to this SSA3 FU is illustrated in Figure 11.11.

$$a \times 23170_{dec} = a \times 10 - 0 - 01010000010 \Rightarrow$$

$$a \times 23170_{dec} = (a \ll 15) - (a \ll 13) + (a \ll 11) + (a \ll 9) + (a \ll 7) + (a \ll 1) \tag{11.3}$$

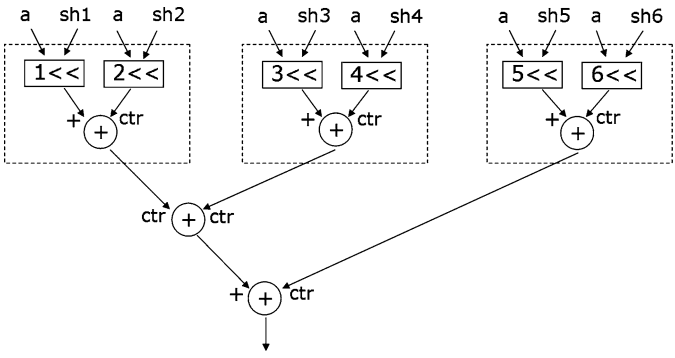


Figure 11.11: SSA3 FU for implementation of multiplication $a \times 23170_{dec}$

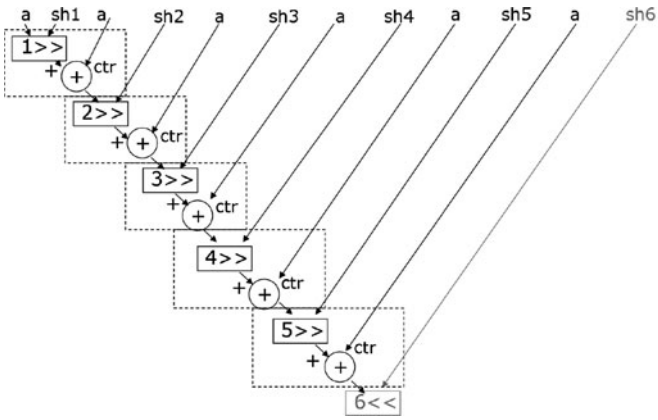


Figure 11.12: SA5 FU for implementation of multiplication $a \times 23170_{dec}$

Then, the multiplication by the worst case constant is converted to a sequence of shift and addition operations considering the right-shift sequential approach, as shown in Eq. 11.4. In this case, five shift-add (SA) FUs are needed, leading to the SA5 FU proposal that is illustrated in Figure 11.12.

$$a \times 23170_{dec} = a \times 10 - 0 - 01010000010 \Rightarrow$$

$$a \times 23170_{dec} = [(((((a \gg 6) + a) \gg 2 + a) \gg 2 + a) \gg 2 + a) \gg 2 + a] \ll 1 \tag{11.4}$$

Observing the implementations of the different approaches of the worst case sequence, a tradeoff is present between the number of cycles needed for execution and the hardware requirements. Due to high number of non-zero digits, transformations to the worst case constant are applied in order to reduce this number and furthermore the cost of implementing the sequence of shift and addition operations. The first transformation applied to the worst constant is the further reduction of its precision. The number of bits used for the representation of the decimal part of all the constants is reduced from 17 to 14 bits precision. The transformation impact has to be verified by a procedure similar to the one used in Section 11.1, namely checking whether the new word-length meets the performance criteria. This is possible due to the fact that the quantization analysis has indicated that more degradation can be accommodated here. After this transformation, the worst case constant is 2896_{dec} and it consists of five non-zero digits, corresponding to four shift operations and four addition operations. The implementation of the new worst case constant on the SSA3 FU is depicted in Figure 11.13 and it is based on Eq. 11.5. The mapping on an alternative option, namely a SA4 FU is depicted in Figure 11.14 and it is based on Eq. 11.6.

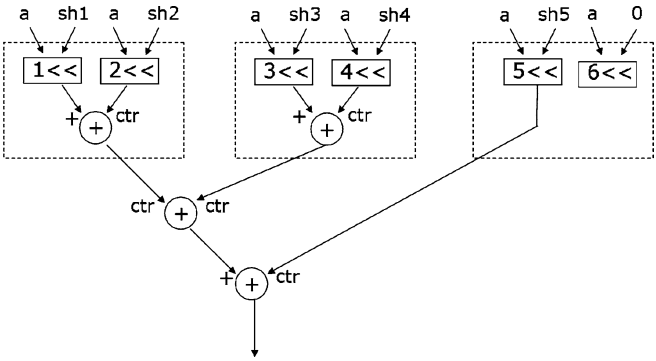


Figure 11.13: SSA3 FU for implementation of multiplication $a \times 2896_{dec}$

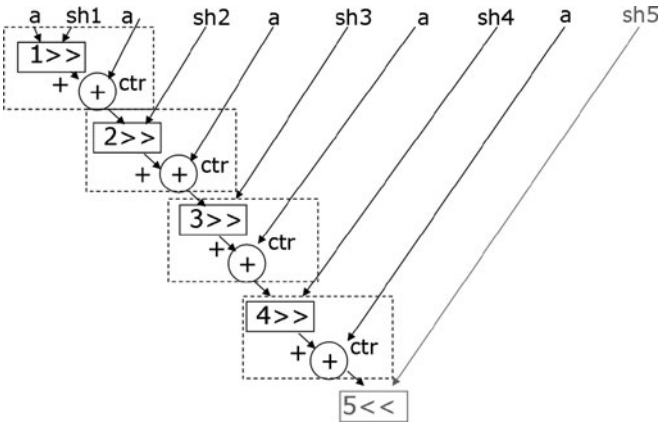


Figure 11.14: SA4 FU for implementation of multiplication $a \times 2896_{dec}$

$$a \times 2896_{dec} = a \times 10 - 0 - 01010000 \Rightarrow$$

$$a \times 2896_{dec} = (a \ll 12) - (a \ll 10) - (a \ll 8) + (a \ll 6) + (a \ll 4) \quad (11.5)$$

$$a \times 2896_{dec} = a \times 10 - 0 - 01010000 \Rightarrow$$

$$a \times 2896_{dec} = [(((((a \gg 2) + a) \gg 2 - a) \gg 2 - a) \gg 2 + a)] \ll 4 \quad (11.6)$$

The utilization of the constants in the multiplications is modified due to the applied transformation, but the worst case constant 2896_{dec} is still dominant. Accordingly, further transformations are applied in order to reduce its number of non-zero digits. The next transformation consists of the replacement of the actual worst case constant with another one, which has a nearby value, but uses less non-zero digits for its representation. Again we have checked whether it meets the overall performance criteria. After the transformation,

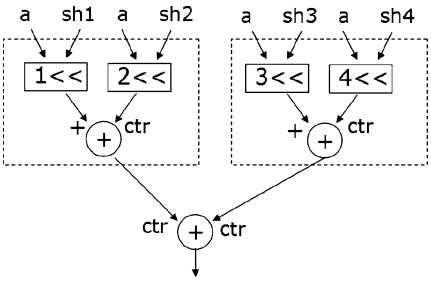


Figure 11.15: SSA2 FU for implementation of multiplication $a \times 2912_{dec}$

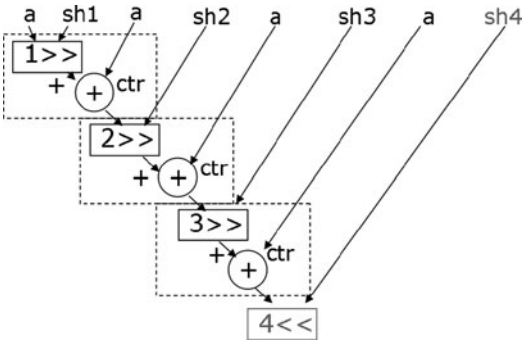


Figure 11.16: SA3 FU for implementation of multiplication $a \times 2912_{dec}$

the new worst case constant is 2912_{dec}. It now consists of four non-zero digits, corresponding to three shift operations and three addition operations. The mapping of the new worst case constant to a SSA2 FU is based on Eq. 11.8 and it is depicted in Figure 11.15, whereas its mapping to a SA3 FU is based on Eq. 11.8 and it is depicted in Figure 11.16.

$$a \times 2912_{dec} = a \times 10 - 00 - 0 - 00000 \Rightarrow$$

$$a \times 2912_{dec} = (a \ll 12) - (a \ll 10) - (a \ll 7) - (a \ll 5) \tag{11.7}$$

$$a \times 2912_{dec} = a \times 10 - 00 - 0 - 00000 \Rightarrow$$

$$a \times 2912_{dec} = [((((-a \gg 2) - a) \gg 3 - a) \gg 2 + a)] \ll 5 \tag{11.8}$$

After the reduction of the non-zero digits of the worst cost constant and its implementation by the parallel and by the sequential with right shifts approaches, the shift factors for every implementation are calculated. The position of the non-zero digits of the CSD coding of the constants specifies the shift factors that must by implemented by the shifters of the SSA FU with the parallel approach, while the difference between the positions of non-zero digits specifies the shift factors that must by implemented by the shifter of the

SA FU with the sequential approach. The mapping of the sequences of shift and addition operations of all the constants to the SSA FU and to the SA FU determines the factors that each shifter has to implement, while the mapping of the worst cost constant determines the maximum number of each FUs in order to execute the sequence in one cycle. It should be stressed again that the SA3 FU stands for three Shift Add Function Units. So the worst constant multiplication is implemented in one cycle, while the SA FU stands for one Shift Add Function Unit, where the worst constant multiplication needs three cycles in order to finish its execution. The use of the parallel approach leads to shift factors that are characterized by a potentially big range, a fact that leads to the need for complex shifters and as a consequence, an increase to the cost and to the energy consumption. In contrast, the shift factors of the right shift sequential approach have similar values. Consequently, the right shift sequential approach is considered for implementation due to the simplicity of its shift factors. The Table 11.1 is summarizing the final shift factors for SSA FU, SA FU and for the SAS FU, which consists of the cascade of the SA FU with one shifter. The SAS FU can potentially be beneficial during the replacement of the constant multiplications of the bioimaging application, since the last shifter can be used for the preparation of the output of the multiplication for the next operation.

In conclusion, the right-shift sequential approach is considered for the remainder of the present case study due to the use of small shift factors. The

Sequential	Sh1	Sh2	Sh3
Detection			
SA3	3.4.5	2.3	2.5
SA2	3.4.5	2.3	
SA	2.3.4.5		
SAS	2.3.4.5	2	
Tracking			
SA3	2.3.5.12	2.3.4	2.4
SA2	2.3.5.12	2.4	
SA	2.3.4.5.12		
SAS	2.3.4.5	2	

Parallel	Sh1	Sh2	Sh3	Sh4
Detection				
SSA2	1.3.4	3.5.6.7	9	10.11
SSA	1.3.4.5.6	7.9.10.11		
Tracking				
SSA2	1.2.3.4.5	3.4.6.7	8.9.10	11.12
SSA	1.2.3.4.5.6.7	8.9.10.11.12		

Table 11.1: Shift factors for the sequential and the parallel approach

right-shift sequential approach is considered for implementation on the SA FU, which uses one Shift-Add FU. It is also used on the SAS FU, which consists of one Shift Add Shift FU. The option of allocating a SASA FU, which consists of two Shift Add Functional Units, is not attractive due to the values of the constants of the bioimaging applications which usually lead to one or three cycles in order to calculate the corresponding multiplication. By combining the proper transformation space (explored in Chapter 10) and the controlled finite-precision degradation of the constant multiplications to sequences of shift and addition operations, it is feasible to heavily reduce the overall implementation complexity.

11.3 Architecture exploration for scalar ASIP-VLIW options

The generic ASIP architecture template used in the COFFEE tool for the exploration of the potential ASIP platform instances, consists of several major components. In particular, it includes a data memory hierarchy with a data memory Level 1 scratchpad DL1 (addressed by a domain-specific DMA engine) and a data foreground memory (which can be either a conventional register-file RF or a very wide register VWR as proposed in Chapter 8). In addition, it has an instruction memory organisation with an instruction memory Level 1 scratchpad (IL1) and a distributed loop buffer LBx (see Chapter 5). Finally, it includes the functional unit (FU) slots and their communication in the ASIP data-path clusters.

For the mapping of the bioimaging application we have used a scalar ASIP version, illustrated in Figure 11.17 and afterward also a data-parallel SIMD version (see Section 11.4). During the architecture exploration with COFFEE, different numbers of data clusters, slots per data cluster, registers in the RFs or words in the VWRs, and positions in the LB are used. The main goal of the architecture exploration is to find the most promising ASIP instances that are characterized by (ultra) low-energy consumption and acceptable area cost, while they maintain or even increase the maximal achievable performance.

The main benchmark used here for the architecture exploration is the fixed point representation of detection algorithm of the bioimaging application (as derived in Section 11.1). The developed methodology can however also be applied to the fixed point representation of the bioimaging tracking algorithm and any other algorithm that meets the characteristics of the target domain. In particular we focus here on data-flow and loop nest dominated applications where the operations are mainly consisting of the add/subtract/shift class including constant multiplications that are then decomposed in shifts and adds. All other operations are assumed to be much less frequent so that

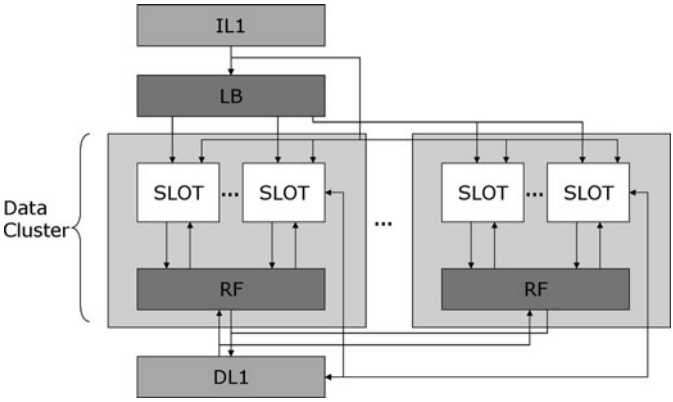


Figure 11.17: Generic architecture used in COFFEE tool

it is feasible to decompose them also in iterative versions (e.g. by means of the very versatile CORDIC algorithm) that typically take as many compute cycles as the output word-length requirement. Algorithms meeting these characteristics are abundant in wireless and bio-sensor processing, but also large subdomains of multi-media or automotive processing belong to this. Hence we cover a quite wide domain with our ultra low-power ASIP-VLIW template.

While mapping the benchmark using the COFFEE tool to architectures with different numbers for data clusters, slots per data cluster, size of the RF size and depth of the LB, the tool set also provides the overall Pareto trade-off curve. This is illustrated for a specific context in Figure 11.18. The curve is normalized to the architecture with the worst energy consumption and to the architecture with the worst performance. The Pareto points of Figure 11.18 indicate architectures that consist of one data cluster, two slots per data cluster, RF size equal to 16 or 64 and LB depth equal to 64 or 128. The execution of the above experiment aims at the reduction of the architecture exploration space, while it gives a first estimate of the energy consumption. The energy estimation of the pareto point 2 is depicted in Figure 11.19.

It should be stressed that a lot of already known and partly well developed methodologies exist for the reduction of the IM energy, the PR (Pipeline Register) energy, the RF energy and the DM energy. The present study is focusing on the further reduction of the processor data path energy. This exploration exploits the use of the constant multiplication conversion to sequences of shift and addition operations (as described in Chapter 10) and then by combining the outcome of the conversion with SoftSIMD data parallelization techniques (as advocated in Section 9.4). The introduction of new FUs impacts not only on the FU energy, but also on the rest energies. Even if the FU energy is shown negligible on the above example, transformations that will be applied will affect the rest of the energies, as it will be shown

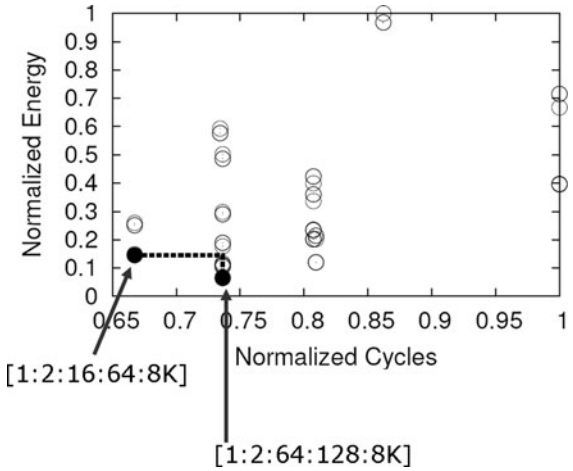


Figure 11.18: Pareto curve of initial architecture exploration. The numbers for each point stand for number of data clusters_number of slots per data cluster_RF size_LB depth

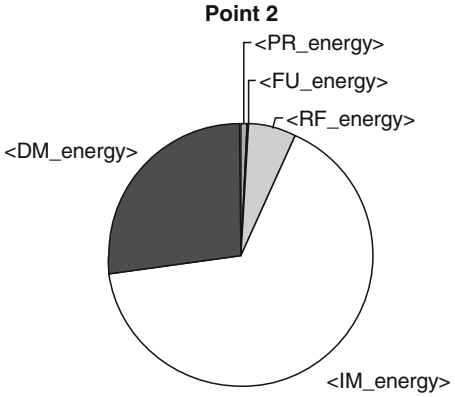


Figure 11.19: Energy pie of point 2, where PR stands for Pipeline Registers, FU for Function Units, RF for Register File, DM for Data Memory and IM for Instruction Memory

later on this chapter. The insertion of new FUs affects the structure of the IMH, and especially the structure of the LB. The IM energy is dominant due to the fact that the LB energy is overestimated. The default LB width is equal to 32 bits per slot and its depth is equal to 128 positions. This LB is oversized though; accordingly, in order to specify the really required IM energy consumption, the actual needed size of the LB has to be determined. Moreover, the insertion of the new FU affects the ISA of the processor and so

also the PL energy. In addition, the RF energy is depending on the RF size, so the minimum needed RF size should be estimated for the biotechnology application. Finally, the DM energy consumption can be reduced by applying methodologies like ADOPT [Mir98] and DTSE [Cat98b, Cat02].

The architectures of the ASIP platform that are considered firstly for exploration are four VLIW architectures which consist of one data cluster and one slot per data cluster (1_1), one data cluster and two slots per data cluster (1_2), two data clusters and one slot per data cluster (2_1), two data clusters and two slots per data cluster (2_2). At the very beginning of the architecture exploration, specific FUs that implement the different approaches of the sequences of shift and addition operations, are inserted to the slots of the different architectures of the ASIP platform. The generic VLIW architecture is characterized by high cost since it includes additional FUs on top of the needed for the efficient implementation of the target application. It is stressed that in the next stages of the architecture exploration, experiments with low-cost ASIPs that have only the needed FUs are carried out. The FUs used in the generic VLIW architectures 1_1 and 1_2 are schematically illustrated in Figures 11.20 and 11.21. It is mentioned that the generic VLIW architectures 2_1 and 2_2 are formed by duplication of the data cluster of 1_1 and 1_2, respectively.

Based on the constant analysis of the bioimaging application, the use of SA and SAS FUs for the implementation of the sequences of shift and addition operations is explored. New intrinsics are inserted to the intrinsic library of COFFEE tool in order to simulate the behavior of these SA and SAS FUs. The constants do not lead to the same execution cycle count, due to the distinct number of shifts and adds in each constant. Hence, different intrinsics with different latencies are introduced for the SA and SAS FU. For example, a constant that represents three shift operations and three addition operations and is implemented on the SA architecture, will require one shift and one addition operation in every cycle. Hence, it then uses the intrinsic SA3 for

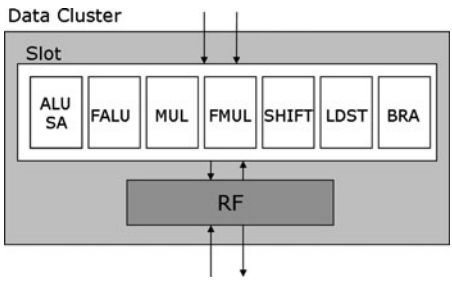


Figure 11.20: Generic VLIW architecture with one data cluster and one slot per data cluster (1_1)

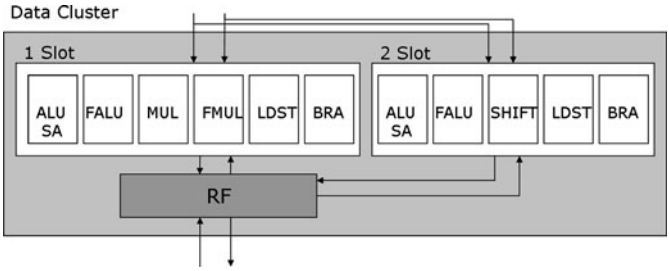


Figure 11.21: Generic VLIW architecture with one data cluster and two slots per data cluster (1_2)

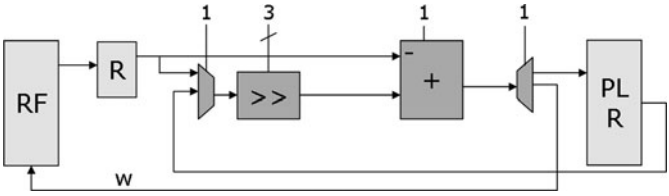


Figure 11.22: SA FU Architecture

which the execution time is equal to three. The same constant when it is implemented on the SAS FU it uses the SAS3 intrinsic for which the execution time is equal to three. The control bits for each FU determine the size of the corresponding loop buffer, which allows the LB energy to be estimated. In this phase of the high-level exploration, the LB energy is overestimated because the detailed mapping is not yet known. That will stay so until the decision of the final ASIP architecture, where the LB size and its energy will be recalculated. That will happen based on actual synthesis results and layout extraction, and that will replace the COFFEE results for the IM energy.

Subsequently, the architectures of the different FUs that implement the sequences of shift and addition operations are analyzed. In Figure 11.22, the architecture of the SA FU is illustrated.

The SA FU consists of one register R, where the multiplicand loaded initially from the RF is kept, or in general it can also contain a variable. In addition, the FU contains one shifter, one adder, one pipeline register PLR, one mux, which selects the input to the shifter, and one demux for bypassing the pipeline register. The pipeline register is used for the constants that need more than one cycle in order to finish the execution of their shift and addition operations. Instead of writing and reading the partial product from the RF, it is fed-back through the pipeline register to the mux and so to the input of the shifter. When the execution of the intrinsic is finished, the result is written back to the RF. The advantage of the use of the pipeline register is based on the fact that, during the execution of the intrinsic, one more register is free

for use, while moreover the non-redundant accesses to the costly RF are avoided. One field of the instruction of the SA FU is the opcode, which consists of one bit for the selection of reading or writing to the RF, one bit for the selection of the mux output, one bit for the selection of the demux output and the needed bits for the selection of the instruction itself. The total needed bits for the opcode are equal to y and they depend on the total number of different instructions that exist in the processor's ISA. The necessary bits for the fields, which determine the operands of the instruction, depend on the RF size and are equal to $2 \times x$ bits, where x is derived from the equation $2 \times x = RF$ size. From the application's constant analysis, the bits that determine the shift factors of the shifter are equal to 2, 3, 4, or 5. Considering a logarithmic shifter, then the needed bit-width for representing the shifter's factors equals to three. One more bit is necessary for the addition or subtraction of the input that is read directly from register R. The instructions that use the SA FU are illustrated in Figure 11.23. It is stressed that the missing bits for the separation of the fields are determined during the architecture exploration.

The SAS FU is illustrated in Figure 11.24 and consists of one register R, where the variable deriving from the RF is kept, two shifters, one adder, one pipeline register PLR, one mux, which selects the input to the first shifter, and one demux, which bypasses the pipeline register.

For the SAS FU, a similar instruction set exploration is performed. The outcome is also similar to what is shown in Figure 11.24 except that now more shift factors and register selection bits have to be concatenated.

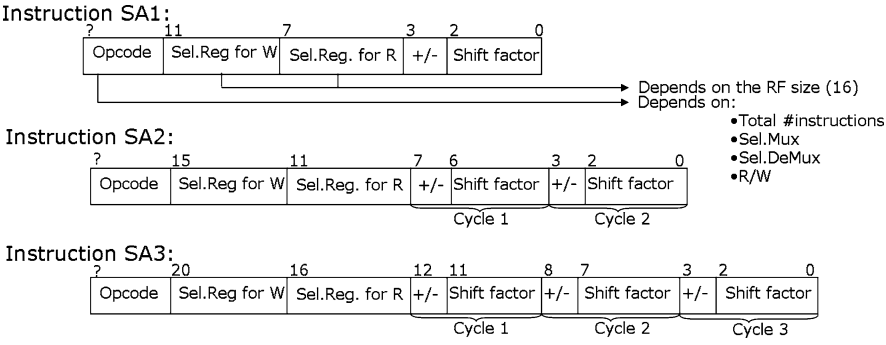


Figure 11.23: Instructions that use SA FU. The instructions are loaded into distributed LB, i.e. LB1 for the RF and LB2 for the FU. The control bits for the addressing of the operands are loaded to the LB1, which is responsible for the memory management, while the control bits for performing the operation to LB2, which is responsible for executing the instruction to the corresponding FU

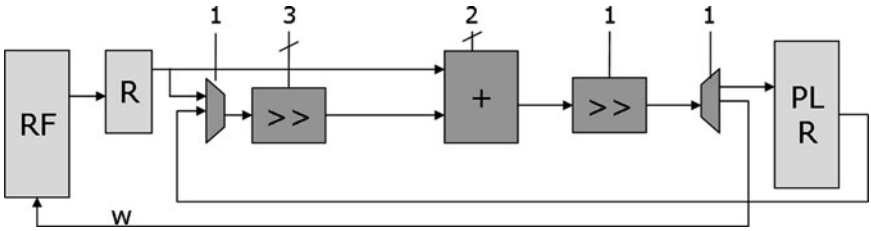


Figure 11.24: SAS FU architecture

The detection algorithm is used as a benchmark for the COFFEE experiments to determine the final ASIP architecture. The critical multiplication loop is defined as the loop that has the maximum number of constant multiplications of the whole biotechnology application and it is the loop that applies the Gauss filter to the frame in the detection algorithm. Firstly an amount of loop transformations is applied to it in order to simplify and reduce the amount of performed operations.

In the original code of the detection algorithm, the Gauss filter is applied through a square 3×3 matrix that holds the Gauss coefficients. In every iteration, the result of the application of the Gauss filter for one pixel is calculated by using the eight neighborhood pixels, which means that each neighbor pixel plus the centre one is multiplied by the appropriate coefficient, the results are summed up and the extracted value replaces the value of the central pixel. The first transformation applied to the critical multiplication loop is the conversion of the square coefficient matrix into two vectors, i.e. the split into two one-dimensional loops. One vector consists of the coefficients, hence it is called coefficient vector. For every pixel it is multiplied by the columns of the 3×3 needed image pixels. The other vector consists of the factors and it is called factor vector. The factor vector is applied after the coefficient vector to the rows of the 3×3 needed image pixels. The factor vector has the appropriate values in order to give equivalent results with the initial ones. The loop nest is split into two loops, where the first loop applies the coefficient vector and the second one applies the factor vector, so the application of the Gauss filter is performed by two loops that iterate through the whole image. The main idea of the next transformation is to merge these two loops into one. Instead of having one loop that iterates through the whole image and that applies the coefficient vector and another loop, which again iterates through the whole image but applies the factor vector, the transformation converts the code into one outer loop that iterates on rows and two inner loops that iterate on columns. Finally, the last loop transformation applied is the merging of the two inner loops. Instead of having two inner loops that iterate on columns, one for multiplication by the coefficient vector and another one for the multiplication be the factor vector, the transformation converts the code with one outer loop and one inner loop. Consequently, while multiplying the

Program 1 Critical Gauss loop after transformations

```
for (x = 1; x < N - 1; x++){
  MulRes0 = imsub[x-1][1]*Gauss[0][0];
  MulRes4 = imsub[x][1]*Gauss[0][1];
  MulRes7 = imsub[x+1][1]*Gauss[0][2];
  imgauss_x[x][1]=MulRes0+MulRes4+MulRes7;
  for (y = 2; y < M - 1; y++) {
    MulRes0=imsub[x-1][y]*Gauss[0][0];
    MulRes4=imsub[x][y]*Gauss[0][1];
    MulRes7=imsub[x+1][y]) * (Gauss[0][2])) ;
    imgauss_x[x][y]=MulRes0+MulRes4+MulRes7;
    MulRes4 =imgauss_x[x][y-1]*3;
    imgauss[x][y-1]=imgauss_x[x][y-2]+MulRes4+imgauss_x[x][y];
    imgauss[x][y-1]=imgauss[x][y-1]>>DECimgauss;
  }
  MulRes4=imgauss_x[x][M-2]*3;
  imgauss[x][M-2]=imgauss_x[x][M-3]+MulRes4+imgauss_x[x][M-1];
  imgauss[x][M-2]= imgauss[x][M-2]>> ]>>DECimgauss;
}
```

pixels by the coefficient vector, the previous pixels can by multiplied by the factor vector, in the same iteration. The final code of the critical Gauss loop is depicted in Program 1.

After the loop transformations, the detection algorithm is used as a benchmark to the COFFEE tool fot the architecture exploration of the ASIP platforms. This phase is divided into three sub-phases, one that maps only the constant multiplications to the specific SA and SAS FUs (Section 11.3.1), one that maps the rest of ALU and shift operations to more generic SA and SAS FUs that are inserted in the generic VLIW architecture of Section 11.3.2 and the final one that makes the final and accurate mapping of the detection in Section 11.3.3

11.3.1 Constant multiplication FU mapping: Specific SA and SAS options

All the constant multiplications that are performed in the detection algorithm of the biotechnology application are mapped to the SA and SAS FUs. It is noted again that, in order for the sequential shift right approach to have the correct results, the input is aligned to the MSB $-$ 1 bit of the word-length of the

output. A new intrinsic is introduced to the COFFEE library in order to apply the correct alignment of the input before the application of the sequence of shift and addition operations. Moreover, sometimes the alignment of the input can be propagated to a previous operation, so no overhead is introduced. Combining the shift factor applied to the previous operation that calculates the input of the constant multiplication with the shift factor that is needed for the alignment, the overhead of adding one more operation is avoided. This is important for the critical Gauss loop that iterates through the whole image. The alignment of the input of the constant multiplication is propagated to the previous operation. Accordingly no overhead is introduced. In addition, a pass of the whole code indicates cases where two shifts can be combined into one. For example, if a variable, when it is computed, is shifted left due to precision and, when it is used, it is shifted right due to alignment for performing correct operation. Then the two shifts can be combined in order the previous operation to prepare the input of next operation. These combinations are applied in the code in order to reduce the number of instructions, and hence the number of operations, since they remove the redundant shift operations. It should be noted that these combinations do not appear in the important loops of the detection algorithm. The COFFEE results for the performance, the energy consumption and the LB depth, are depicted in Figures 11.25–11.27.

It is stressed, once again, that the energy should be used only for a crude relative comparison and that the performance of the SA and SAS FUs is approximated. The IM energy shown in Figure 11.28, is somewhat overestimated due to the upper bound taken for the LB energy.

11.3.2 FUs for the Generic SAs

This option explores a more generic SA FU, which is called GSA FU, and a more generic SAS FU, which is called GSAS FU. In the GSA and GSAS FUs not only the constant multiplications can be mapped, but also the rest of

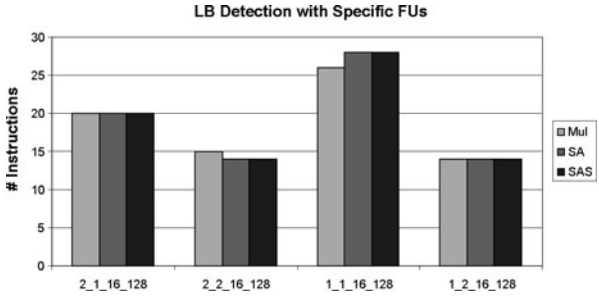


Figure 11.25: LB depth comparison of the detection algorithm for the implementation with the multiplier FU and the SA and SAS FU

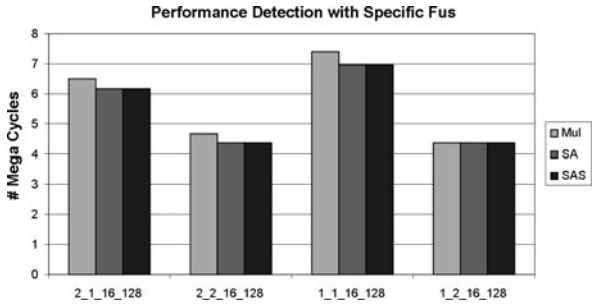


Figure 11.26: Performance comparison of the detection algorithm for the implementation with the multiplier FU and with the SA and SAS FU

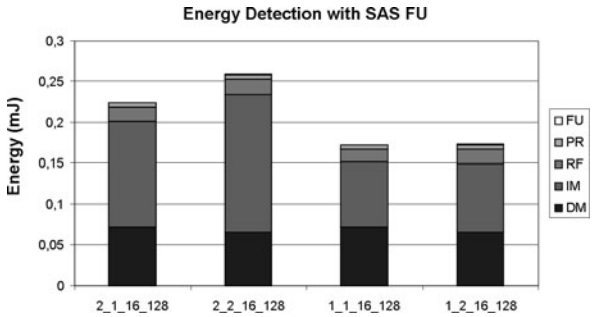


Figure 11.27: Energy comparison of the detection algorithm for the implementation with the multiplier FU and with the SA and SAS FU

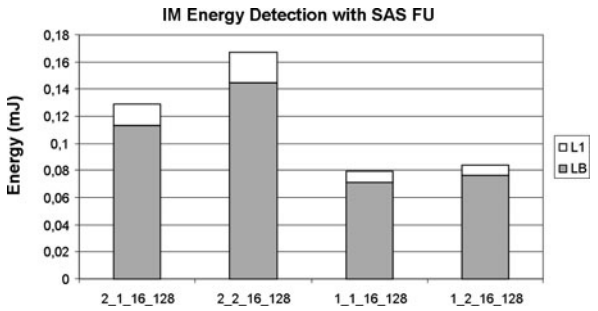


Figure 11.28: IMH energy comparison of the detection algorithm for the implementation with the multiplier FU and with the SA and SAS FU

the ALU and shift operations that would be required for the detection or tracking modules. For instance, the GSA and GSAS FUs are able to use not only right shift operations but also left shift operations. For the GSA and GSAS FUs new intrinsics are introduced to the COFFEE tool, as illustrated in the Figures 11.29 and 11.30. The needed bits for selecting a register, e.g. operands from the RF, equal four since the RF size is now set to 16 registers. That is needed due to the increased set of operands and life-times in the part of the code that is not focused only on the Gauss filtering.

It should be mentioned that the GSA and GSAS data-path can be used to execute pure shifting. In this case, the adder adds a zero value to the output of the shifter. This zero value could come from a register, which has as disadvantage of one redundant access to the RF for reading the zero value. Moreover, one less register is available for the other variables. A better solution can be achieved either by bypassing the adder using one demux and one mux, or by driving the first input of the adder using an AND gate. In case of performing the addition with the zero, then the control bit of the AND gate is set to zero, forcing the output to have a zero value. That is probably the most interesting option but it depends on the technology and the cell library used. Finally, another solution can be enabling of the register reset signal after its use.

The code of the detection algorithm is converted using the new intrinsics for the GSA FU and the GSAS FU, and combining, whenever is possible, the shift applied to a next operation with the shift of the previous operation. These combinations reduce the number of operations since they remove the redundant shift operations. Moreover, the shift operations are mapped to the shifter of the GSA FU and the GSAS FU by adding a zero to the output of the shifter, while the addition or subtraction operations are mapped to the adder of the

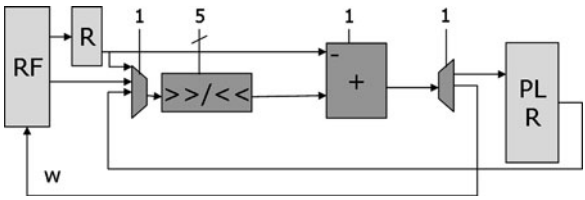


Figure 11.29: GSA FU architecture

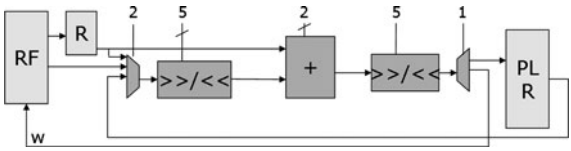


Figure 11.30: GSAS FU Architecture

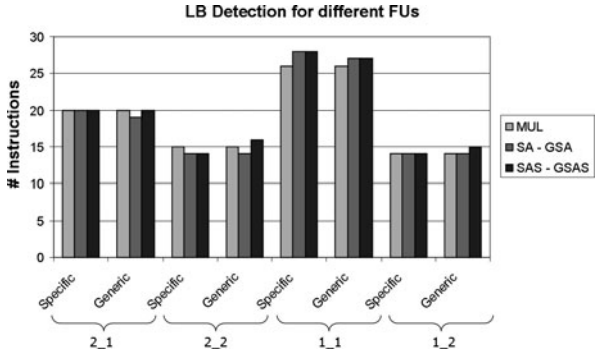


Figure 11.31: LB depth comparison of the detection algorithm that uses FUs only for implementing the constant multiplications and of the detection algorithm that uses more generic FUs for mapping the whole application

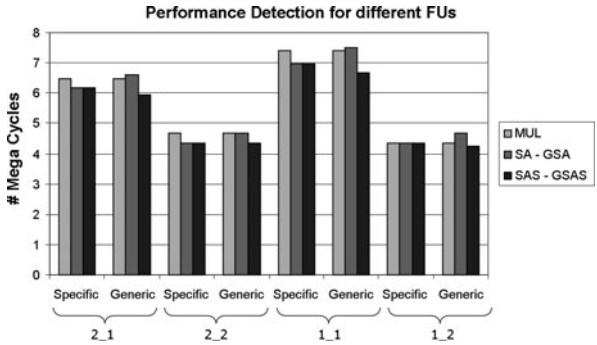
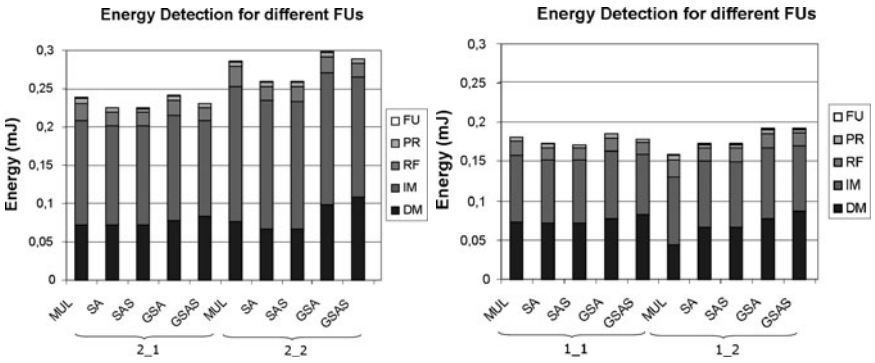


Figure 11.32: Performance comparison of the detection algorithm that uses FUs only for implementing the constant multiplications and of the detection algorithm that uses more generic FUs for mapping the whole application

GSAS FU and GSAS FU by shifting zero positions. The COFFEE results for the LB depth, the performance and the energy consumption between the implementations using the specific SA and SAS FUs and the generic GSA and GSAS FUs are depicted in Figures 11.31–11.33.

Based on the above results, the LB depth is set to 32 positions for reducing the IM energy consumption of the COFFEE results. After the selection of the final ASIP architecture, the LB size will be explicitly defined and the IM energy will be recalculated. Considering that 8 Mega cycles are available for performing both the detection and the tracking algorithm between two consecutive frames, only the generic VLIW architectures, which accommodate more ILP while avoiding the intercluster copy instructions, are further



(a) Architectures 2_1 and 2_2 (b) Architectures 1_1 and 1_2

Figure 11.33: Energy comparison of the detection algorithm that uses FUs only for implementing the constant multiplications and of the detection algorithm that uses more generic FUs for mapping the whole application

explored in order to meet the performance requirements of the bioimaging application. Furthermore, the GSA and GSAS FUs are considered due to the fact that they reduce the cost of the architecture since the mapping of the rest of the operations to these FUs increases their utilization.

11.3.3 Cost-effective mapping of detection algorithm

The architectures that are considered for further exploration consist of the ones that provide sufficient parallelism, e.g. the architecture with one data cluster and two slots (1_2) and the architecture with two data clusters and two slots (2_2). The next step consists of removing the redundant FUs of the VLIW architectures in order to reduce their cost and to create thin ASIP architectures. The thin ASIP architecture for the 1_2 case is illustrated in Figure 11.34, while the thin ASIP architecture 2_2 is formed by duplication of the data cluster of the thin ASIP architecture 1_2.

However, in order to select the final ASIP architecture, a more accurate modeling and more cost-effective mapping of the detection algorithm is needed. The more detailed performance and energy consumption models for the GSA and GSAS FUs are used, while the unconverted operations of the benchmark are modeled through intrinsics. We have also converted the trigonometric functions, square roots and divisions of the code of the biotechnology application to sequences of shift and addition operations, but the details are not incorporated in this book chapter. The trigonometric and the square root functions can be converted with a CORDIC implementa-

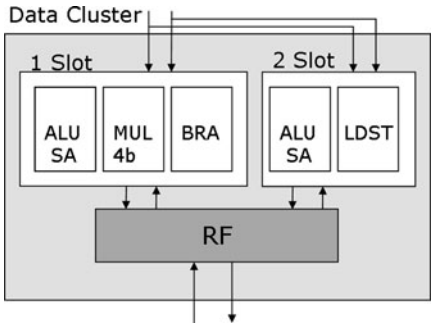


Figure 11.34: Thin ASIP architecture with one data cluster and two slots per data cluster(1_2)

tion [Maz93, Vol59], whereas the division can be achieved also with iterative methods based on shifts and additions. New intrinsics that simulate the behavior of the above operations are introduced to COFFEE tool for the correct modeling of the benchmark. In order to have a good approximation of the needed execution time, the trigonometric and square root functions have an execution time that is linearly proportional to their output word-length, whereas the execution time of the division is equal to the word-length of the denominator plus one. Furthermore, no pipeline is allowed for either the GSA and the GSAS FUs or for the new intrinsics. The next two applied steps consist of further code optimizations focusing on the operations and FUs. The variable multiplications are converted using multiple precision multipliers of 4 bits [Pau92, Pau95]. In this case, the longer multiplication size is modeled by breaking up the operands into 4 bit segments and then accumulating the 4 by 4 bit partial multiplier results. When we start initially with a 16 by 16 bit multiplication, we end up with 16 partial multiplier results to be added up with the appropriate alignment shifts. Their behavior is again introduced through the COFFEE tool using new intrinsics, each one with the appropriate execution time depending on the word-length of the two inputs of the multiplication. Moreover, the multiplications applied to the indexes of the arrays are pre-calculated and the results are stored in a Look Up Table (LUT). During the execution of the detection algorithm, the values of the multiplications arederiving directly from the LUT. In addition, the loop that applies the conversion to the binary image and the loop that calculates the least square methods are merged in order to remove unnecessary if statements.

The transformed detection algorithm is mapped through the COFFEE tool to the FAT (Generic VLIW) and to the THIN (ASIP) architectures using the GSA FU and the GSAS FU. The results for the LB depth, the performance and the energy are compared in Figures 11.35–11.37.

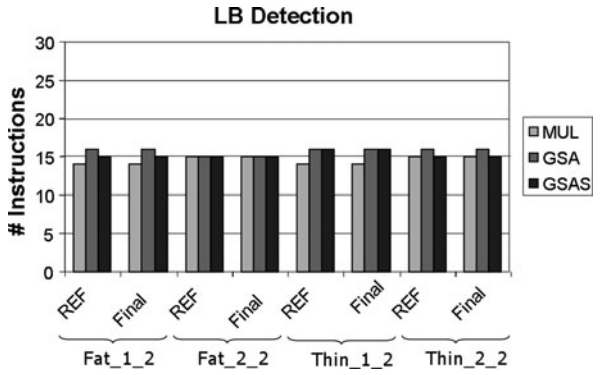


Figure 11.35: LB depth comparison of the detection algorithm with and without the loop optimizations for the implementation with the multiplier FU and with the GSA and GSAS FU

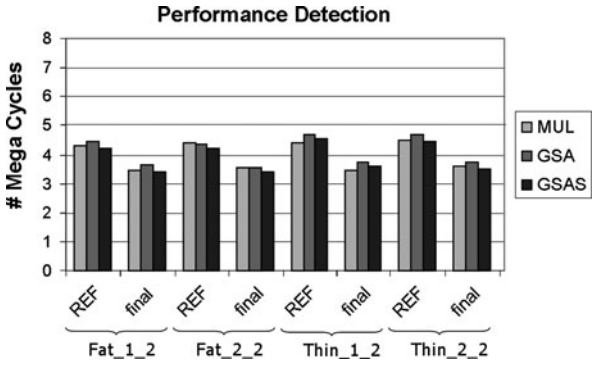


Figure 11.36: Performance comparison of the detection algorithm with and without the loop optimizations for the implementation with the multiplier FU and with the GSA and GSAS FU

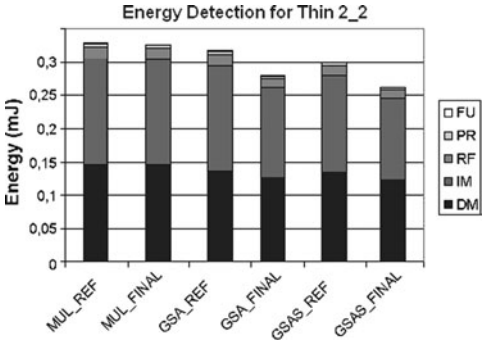


Figure 11.37: Energy comparison of the reference and final detection algorithm for the implementation with the multiplier FU, GSA FU and GSAS FU

11.4 Data-path architecture exploration for data-parallel ASIP options

The Single Instruction Multiple Data (SIMD) instructions exploit the data parallelism that is contained in applications and increase their performance, since they execute one instruction on different data at the same time. In addition to the conventional hardware-enabled SIMD approach, we have also exploited the software-SIMD concept (see Chapter 9). The Soft-SIMD concept is applied especially on the critical Gauss filtering loop of the detection algorithm. The approach used for the required correction operations consist of combining the worst case sub-word and one shuffler. Instead of using always the worst-case sub-word, an intermediate sub-word can be used. When even more space is required, repacking operations are applied to the already packed words by a shuffler. The analysis of the operations performed on the critical Gauss loop of the detection algorithm determines the intermediate sub-word that should be used for the packing of the data. It should be mentioned that the loop should be split into two loops, one that uses Soft-SIMD over the x iterator and another that uses Soft-SIMD over y. Otherwise the SIMD directions are not compatible. For the correct modeling of the benchmark, the GSAS intrinsics are used because the shift operations performed by the second shifter can be performed by the shuffler in one cycle. Moreover, a new intrinsic is inserted to the COFFEE simulation for modeling the repacking operations performed by the shuffler.

The proposed architecture GSAS FU (Generic Shift Add Shuffle Function Unit) for the implementation of Soft-SIMD consists of one shifter, one adder, one shuffler and four registers. The R1 register is used for keeping constant the packed word that is read from the VWR (Very Wide Register). The R2 and the R3 registers are used for storing the packed words required for the repacking operations. The R4 is used for feed back from the shifter and the adder. The VWR is able of storing an amount of packed words, in this case 12, as illustrated in Figure 11.38.

The loop body of the sequential code of the critical Gauss loop of the detection algorithm, i.e. the Gauss loop, is illustrated in Program 2.

Subsequently, the mapping of the critical Gauss loop to the proposed architecture is performed. The first operation performed is the addition of two image pixels (imsub) using the adder of the GSAS FU (Generic Shift Add Shuffle Function Unit). Based on the quantization results (Section 11.1), the word-length of the imsub variable is equal to 7 bits. The imsub variable is packed in 7-bit sub-words in memory. After performing the addition, the result needs a word-length equal to 8 bits. Therefore, the input variable imsub is repacked to 8-bit sub-words with the use of the shuffler FU in order to avoid a later repack operation, as illustrated in Figure 11.39.

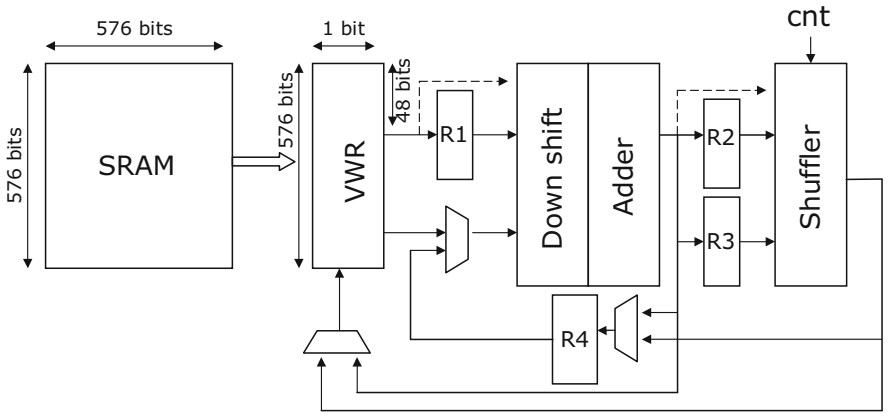


Figure 11.38: Architecture for mapping critical Gauss loop implemented with sequences of shift and addition operations using Soft-SIMD

Program 2 Sequential critical Gauss filter loop implemented with sequences of shift and addition operations

```
for (x = 1; x < N - 1; x++){
  for (y = 2; y < M - 1; y++) {
7   MulRes0 =          GSAS1(imsub[x-1][y],imsub[x+1][y],0x0,0x0);
8   MulRes7 =          GSAS3(MulRes0,MulRes0,0x8018040,0x20);
9   MulRes4 =          GSAS3(imsub[x][y],imsub[x][y],0x60100A0,0x34);
10  imgauss_x[x][y] = GSA1 (MulRes7,MulRes4,0x20,0x0);
11  MulRes2 =          GSAS1(imgauss_x[x][y-1],imgauss_x[x][y-1],
                       0x20,0x0);
12  MulRes5 =          GSAS1(imgauss_x[x][y-2],imgauss_x[x][y],0x0,0x0);
13  imgauss[x][y-1] = GSA1 (MulRes5,MulRes2,0x24,0x0);
  }
}
```

Figure 11.39: Addition of 8 bit sub-words

The second operation is performed on the output of the addition and represents the implementation of the multiplication of the constant 655_{dec} using the GSAS FU for 3 cycles. The input has word-length equal to 8 bits while the output of the multiplication needs 9-bit word-length. Due to code transformations that have been applied to the Gauss loop, the appropriate

information about the required intermediate word-lengths is not available. Therefore, a safe but conservative assumption that the output needs max $(word\text{-}length(input1))$,$word\text{-}length(input2) = max(8, 13)$ is made. No need exists for guard bits (see Section 9.4.2) due to the known dynamic range of the variables. hence the word-length of the output is set equal to 12 bits. Analyzing the steps performed during the execution of the sequence of shift and addition operations, the worst-case subword is equal to 17 bits. For the rest of the analysis it is assumed that the decision of 17 bits is too conservative, so 16 bits are used. That is motivated because the final resolution of results that are passed to the y loop is only 13 bits. In order to correctly apply the Soft-SIMD, the input needs to be packed in 12 sub-words.

During the first cycle, the partial product is shifted four positions to the right and is still inside the sub-word. The second shift operation passes the partial product outside the sub-word. This indicates that repacking through the shuffler FU of the partial product must be performed from 12-bits to 16-bit subwords. Hence also the input repacking must be performed from 12-bits to 16-bit subwords. The second and the third cycle are performed on 16-bit subwords and at the end of the third cycle the result is repacked at 12-bits subwords using the shuffler FU. The next operation is performed on the imsub variable, which is packed in 7-bit sub-words in memory. This represents the implementation of the multiplication of the constant 983_{dec} using the GSAS FU for 3 cycles. The input has word-length equal to 7 bits while the output of the multiplication requires a 9-bit word-length. As already mentioned, due to code transformations that have been applied to the Gauss loop, the appropriate intermediate word-lengths are not fully known here yet so a safe assumption that the output is equal to 12-bit word-length has been made. Therefore, the input variable is packed to 12-bit sub-words. This could have been further optimized by locally applying the quantisation optimisation of Section 11.1 again.

We will now explain in more detail how this 983_{dec} constant multiplication is decomposed into shift and add operations. We will also indicate how the guard bits for each of these subword operations is performed. Subword manipulation is happening within the arithmic operation of the SIMD operations and during repositioning of bits inside the word. That leads to moves and extensions to the left and right, and those risk to overwrite bits that belong to other neighbouring subwords. To avoid this problem, guard bits are introduced at the sides of the subword as buffers.

Table 11.2 depicts the form in which the 12-bit input variable is packed. During the first cycle of the multiplication, it is used both as multiplicand and as partial product. It consists of the 7-bit actual wordlength of the imsub variable, preceded by one guard bit that is needed because of the addition during the first cycle. It is extended by 4 guard bits at the right-hand side,

Multiplicand	-	g	x	x	x	x	x	x	x	g	g	g	g	-	-	-
Partial result	-	g	x	x	x	x	x	x	x	g	g	g	g	-	-	-
>> 3	-	g	g	g	g	x	x	x	x	x	x	x	g	-	-	-
+ imsub[x][y]	-	x	x	x	x	x	x	x	x	x	x	x	g	-	-	-

Table 11.2: First sequence of shift and addition operations due to multiplication M4 by constant 983_{dec} to 12 bit subwords

Multiplicand	g	g	x	x	x	x	x	x	x	g	g	g	g	g	g	g
Partial result	g	x	x	x	x	x	x	x	x	x	x	x	g	g	g	g
>> 2	g	g	g	x	x	x	x	x	x	x	x	x	x	x	g	g
+ imsub[x][y]	g	x	x	x	x	x	x	x	x	x	x	x	x	x	g	g

Table 11.3: Second sequence of shift and addition operations due to multiplication M4 by constant 983_{dec} to 12-bit subwords

that guarantee that during the first shift to the right by 3, no bits are shifted out of the subword toward its neighbour. In this illustration, the guard bits are represented by “g”, while “x” stands for the different data bit values and the double line indicates the position of the decimal point.

The second and third shift operations pass the partial product outside the subword. They indicate that repacking must be performed from 12-bit to 16-bit subwords through the shuffler FU of the partial product. Hence the input has to be repacked, before entering the second cycle of the multiplication. In this case, although only one guard bit preceding the actual value of the multiplicand would be enough to prevent the final result from possible overflow during the two next additions of the multiplication, one additional issue needs to be taken into account. The final result of this multiplication, namely the variable MulRes4, participates in an addition that represents the fourth operation of the loop body of the critical Gauss loop in Program 2. This implies that the result of this multiplication must already be provided with a guard bit, that will ensure the correct execution of the subsequent addition. Thus, an extra guard bit is needed at the left-hand side of the partial result subwords, which leads to two guard bits required at the left-hand side of the repacked multiplicand. These considerations lead to the choices shown in Table 11.3 regarding the 16-bit representation of the multiplicand and the partial result.

During the third cycle, the partial result needs to be shifted by 5 (Table 11.4). Because of the 16-bit length of the partial result, 3 bits have to be shifted out

Multiplicand	g	g	x	x	x	x	x	x	x	g	g	g	g	g	g	g
Partial result	g	x	x	x	x	x	x	x	x	x	x	x	x	x	g	g
$>> 5$	g	g	g	g	g	g	x	x	x	x	x	x	x	x	x	x
+ imsub[x][y]	g	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Result:MulRes4	g	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Table 11.4: Third sequence of shift and addition operations due to multiplication M4 by constant 983_{dec} to 12-bit subwords

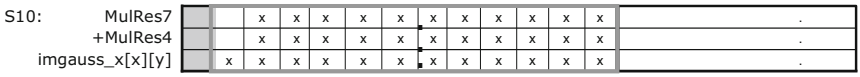


Figure 11.40: Addition of 12-bit sub-words

of the subword. In this case, care must be taken so that the extra bits are "cut off" and do not enter the neighbouring subword. The output is repacked to 12-bit subwords.

The last operation that is performed while using Soft-SIMD on x is the addition of the results of the constant multiplications by the adder of the GSAS FU. Both the inputs are packed in 12-bit subwords. They are aligned, so that the addition can be performed without repacking of the variables. This is illustrated in Figure 11.40.

It should be mentioned that shift operations for alignment of the decimal points are also performed during the repacking operations. The rest of the Gauss body loop consists of operations that are executed with Soft-SIMD over the y iterator. Between the operations performed using Soft-SIMD in x and Soft-SIMD on y, the output variables of Soft-SIMD on x are passed through the shuffler FU in order to be packed in 13-bit sub-words and then they are stored in the VWR and from there to the data memory (see data layout in Figure 11.41). All the operations using Soft-SIMD on y can be performed without need for repacking since the results are always inside the 13-bit sub-words, as illustrated in Figure 11.42.

Applying the above analysis on the Gauss body loop, the loop is converted to the one depicted in Program 3. It is noted that the data path width is set equal to 48 bits, so the packed data with 8-bit sub-words consist of six different data, the packed data with 12-bit sub-words consist of four different data and the packed data with 16-bit sub-words consist of three different data. In order to have full use of the packed bits using different sub-word each

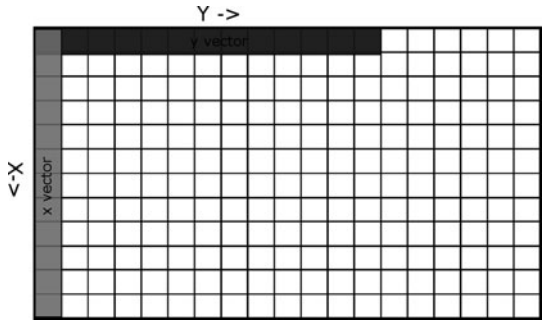


Figure 11.41: Packed words on x vector and on y vector in memory

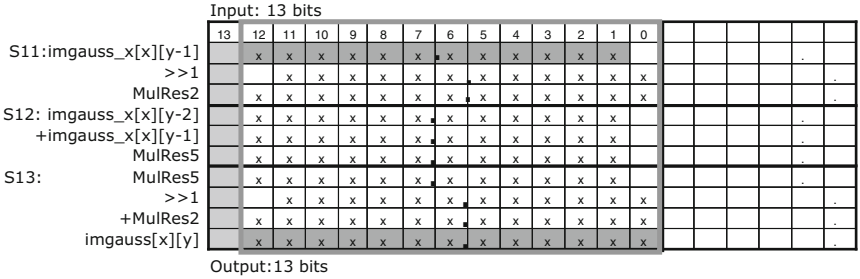


Figure 11.42: Shift operations and additions performed on the y vector on 13-bit sub-words

time, the code must be partially unrolled, as illustrated on the example of Figure 11.43.

The operations performed on 8-bit sub-words are unrolled once and the operations performed on 12-bit sub-words are unrolled twice in order to create 12 different data that completely full each packed word. The code in the Program 3 illustrates the converted Gauss loop in which Soft-SIMD on x and Soft-SIMD on y have been applied.

The correct modeling of the Soft-SIMD through the COFFEE tool is performed by unrolling the operations that need more that one cycle. This is done due to the need for repacking between the cycles that execute the sequence of shift and addition operations, especially when the shifting factor pushes the sub-word out of its available data-path width. This fact indicates that one cycle is required for executing the shift and addition operation and another cycle for the repacking operation through the shuffler. Furthermore, it should be stressed that cases are present in which the shifter, the adder and the shuffler can be executed in one cycle by bypassing the pipeline registers before

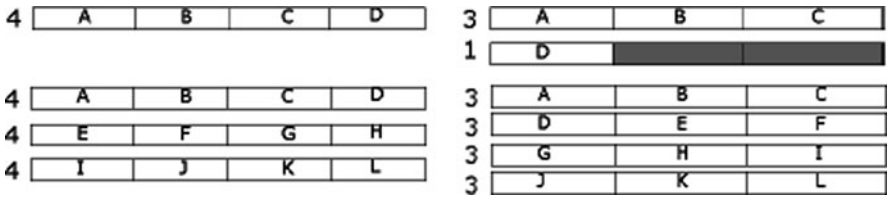


Figure 11.43: Unrolling of operations for full utilization of packed words

Program 3 Parallel critical Gauss loop using Soft-SIMD on x and on y

```
for (x = 1; x < N - 1; x=x+12){
  for (y = 2; y < M - 1; y++) {
    MulRes0_8_a = GSAS1(imsub_8[x-1][y],imsub_8[x+1][y],0x0,0x0);
    MulRes0_8_b = GSAS1(imsub_8[x-5][y],imsub_8[x+7][y],0x0,0x0);
    MulRes0_12_a = REPACK(MulRes0_8_a,0);
    MulRes0_12_b = REPACK(MulRes0_8_a,MulRes0_8_b);
    MulRes0_12_c = REPACK(MulRes0_8_b,0);
    MulRes7_12_a = GSAS3(MulRes0_12_a,MulRes0_12_a,0x8018040,0x20);
    MulRes7_12_b = GSAS3(MulRes0_12_b,MulRes0_12_b,0x8018040,0x20);
    MulRes7_12_c = GSAS3(MulRes0_12_c,MulRes0_12_c,0x8018040,0x20);
    imsub_12_a = REPACK(imsub_8[x][y],0);
    imsub_12_b = REPACK(imsub_8[x][y],imsub_8[x+6][y],2);
    imsub_12_c = REPACK(imsub_8[x+6][y],0);
    MulRes4_12_a = GSAS3(imsub_12_a,imsub_12_a,0x60100A0,0x34);
    MulRes4_12_b = GSAS3(imsub_12_b,imsub_12_b,0x60100A0,0x34);
    MulRes4_12_c = GSAS3(imsub_12_c,imsub_12_c,0x60100A0,0x34);
    imgauss_x[x][y] = GSAS1(MulRes7_12_a,MulRes4_12_a,0x20,0x0);
    imgauss_x[x+12][y] = GSAS1(MulRes7_12_b,MulRes4_12_b,0x20,0x0);
    imgauss_x[x+24][y] = GSAS1(MulRes7_12_c,MulRes4_12_c,0x20,0x0);
  }
}
for (x = 1; x < N - 1; x++){
  for (y = 2; y < M - 1; y=y+3) {
    MulRes2 = GSAS1(imgauss_x[x][y-1],imgauss_x[x][y-1],0x20,0x0) ;
    MulRes5 = GSAS1(imgauss_x[x][y-2],imgauss_x[x][y],0x0,0x0 );
    imgauss[x][y-1] = GSAS1(MulRes5,MulRes2,0x24,0x0);
  }
}
```

the shuffler. The transformed Gauss loop using Soft-SIMD is embedded in the program of Figure 11.46, which will be described further on.

The performance obtained by the COFFEE scheduler for the Gauss loop and the architecture with one data cluster and two slots is illustrated in Figure 11.44.

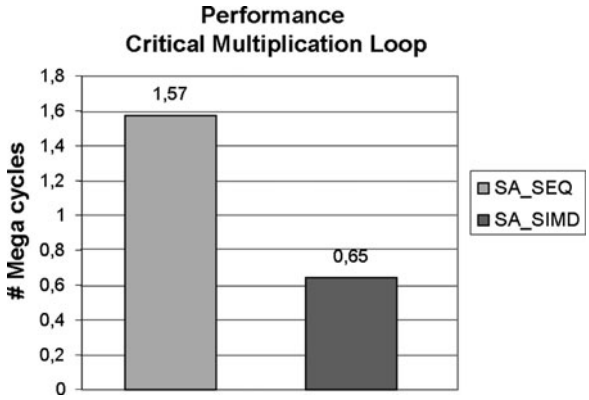


Figure 11.44: Performance comparison of the sequential critical Gauss loop and of the parallel critical Gauss loop using Soft-SIMD, both implemented with sequences of shift and addition operations

11.5 Background and foreground memory organisation for SoftSIMD ASIP

11.5.1 Basic proposal for 2D array access scheme

As motivated above, we would like to have a scheme where the data-path is combining a number of subwords into a word, of, e.g. 48 bit. And then for using the VWR based foreground memory, we need an SRAM interface of e.g. 576 bit (12 times wider), i.e. with a complete "line". That is one of the key components of our proposed scheme, see also Figure 11.45. See Section 11.4 and [Kri09] for more motivation for the actual widths that are used in the subword parallel ASIP data-path organisation. For all practical illustrations in this section we will utilize these same numbers.

However, in between the data-path and the background memory, we also need to plug a foreground memory as a "buffer" to compensate for the unequal read and write access sequences. Conventional multi-port register-files will not provide a really low-energy access, so instead we have proposed to utilize very wide registers (VWRs) (as described in Chapter 8 and [Rag07a]) of, e.g. 576-bit wide, that have 1-port cells but multiple external ports of e.g. 48 bit to the data-path. This asymmetrical interface allows to convert the relative small size of the data-path effectively to a very wide SRAM interface. The VWR can be pitch-matched to the output lines of the SRAM, and can even replace the conventional in/out buffers, allowing to save even more energy!

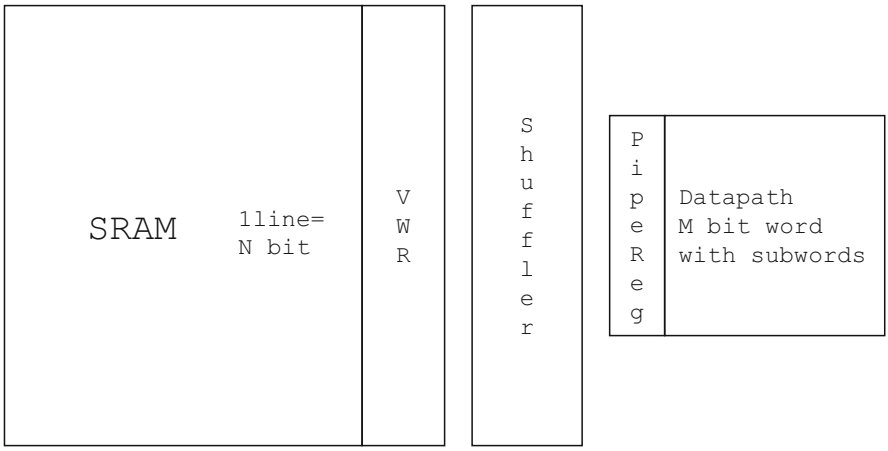


Figure 11.45: Main option for background and foreground storage organisation

Finally, we also need an operator to reorganize the words in the line and potentially also the subwords inside a word. For this purpose, we can reuse the shuffler that is designed to reorganize the subwords in the word (see Figure 11.38). An efficient and flexible design of such a versatile shuffler is presented in [Rag07b, Rag07c]. However, then we would require a 576-bit wide shuffler which is quite expensive in area and energy consumption. So instead, in our main option, we will use a (sub)word selection network that only contains some simple multiplexers. Indeed, only a very small subset of all 576 to 576 crossbar connections are needed here. That subsetting is enabled in the design of [Rag07c]. In practice, we will only exploit the fully regular and relatively "local" reshuffles. Those are namely sufficient for our purpose here. In the illustration further on we need only a maximal shuffling scope of 72 bit out of the 576-bit total width.

All the more complex subword and bit manipulations (such as realignment, addition of guard bits, left/right shifting) then have to be performed on the regular (48-bit) shuffler that belongs to the last section of the data-path proper.

We assume that the (sub)word selection network is positioned next to the VWR, in a pitch-matched way. That selection network is then also taking care of the selection of specific VWR words to the data-path itself. Typically the data-path will then also contain an M-bit wide local register to buffer the required data. That will reduce the requirement on most energy-costly VWR accesses whenever temporal data reuse can be exploited.

Even with this communication network in place, it is however not yet possible to correctly and effectively exchange data-parallel (sub)words between the SRAM background memory and the VWR plus datapath. Indeed, we need a hardware enabling for writing words both in the conventional way along the rows (576-bit wide) but in addition also along part of the column direction (e.g. 72-bit wide).

The most promising and energy-efficient options to realize the above have been described in [Cat09]. They are not the focus of this book. The option with the best energy-efficiency requires a modification in the periphery of the SRAM, namely in the write buffer and global address decoder for the global bit-line selection. We do not want to modify the internal cells and the local bit line (LBL) organisation [Cos08] because that would require a complete redesign and recharacterisation. Our proposal hence does not require any changes in the SRAM matrix itself and thus the design overhead is relatively small. Moreover, the energy gain compared to a conventional RAM scratch-pad is significant, as demonstrated in [Cat09]. That report also describes the schedule for the full data exchange in the Gauss loops between the SRAM background memory and the VWR plus datapath.

To complete this subsection, also the alternative for a conventional SRAM without modified write organisation will be sketched. In that case, everything related to the horizontal/vertical "transpose" reorganisation should be taken care of either by the SRAM addressing, or by the regular datapath shuffler. When we try to use the addressing, the limitation of the SRAM data layout (that has to be matched between the write and the read phase) will force us to pay a very heavy penalty in cycles, and hence also energy. So that option is not really attractive. Then we can as well not exploit the data parallelism in that part of the algorithm which exhibits the described data access to more dimensional arrays! In particular, for the 3×3 matrix operation in the bioimaging detection module it would mean that we go back to 8-bit accesses to the SRAM for the read and write operations. A hybrid would then still be feasible where we write with individual 8-bit subwords (at the proper locations) and then we can read in a data-parallel way from the SRAM into the VWR. For the schedule above, this would mean we have $9 \times 8 = 72$ write accesses per line instead of 9 writes in the fully optimized scheme of [Cat09] and the single read access of 576 bit remains the same. Overall we then have 73 writes and 2 reads instead of 10 writes and 2 reads. The SRAM energy will go up with approximately a factor 6.25 then. That makes the SRAM related energy become quite high again in the overall energy pie. Still, in this hybrid way it is slightly lower than the total data-path contribution.

When we try to use the shuffler, the performance penalty will reduce, especially depending on how wide the scope and flexibility of the allowed shuffling really is. But that improvement in performance goes fully at the cost of a significant energy increase. Even with small scopes, the shuffler is already an

energy bottleneck, due to its dependence on many and relatively long interconnections. Obviously, also this option is not that attractive. It has to be carefully evaluated up-front whether enough can be gained overall with the data-parallel operation compared to the energy penalty paid in the shuffling.

Finally, we can also try to use non-conventional addressing schemes. An example of this is a special type of zig-zag (Morton type) addressing that is utilized, e.g. in multi-dimensional discrete wavelet schemes (see, e.g. [Laf99]). Then we can probably reach a better compromise between performance and energy penalty. But the "space-filling scan" puts higher demands on the AGU (resulting in energy penalties there) and it will still imply a significantly higher overall energy overhead compared to the proposal defined here. In summary, the conventional SRAM memory module is not an attractive option here. So the domain-specific data-path of the ASIP is best extended with the domain-specific SRAM module of [Cat09].

Note also that the proposed two-dimensional background memory access scheme is quite generic for all data-parallel (including soft-SIMD) contexts, and it is also fully compatible with the other ultra-low energy optimisations that have been described earlier in this chapter. In particular, it enables the effective use of customized data word-lengths (e.g. 6, 8, 12, 16, 24 bit), shift-add based versions of constant multiplies, of division, of trigonometric functions and so on.

11.5.2 Overall schedule for SoftSIMD option

The bioimaging application code of Program 3, after applying the appropriate unrolling factor described above, can be used as basis for an extremely dense and energy-efficient mapping effort. When we exploit the full functionality and flexibility of all the operators in the Soft-SIMD data-path and the VWR to the limit, this leads to a relatively complex and entangled schedule which is highly optimized. This is described in the program of Figure 11.46, which is based on [Psy10].

Figure 11.38 is used as the processor architecture instance on which we map this program. The Shift-Add unit is fully occupied for each of the 27 cycles and that forms the critical path. The schedule in Figure 11.46 contains only the first 17 cycles out of a total of 27 in order not to overload the description here. The more energy-expensive shuffler unit is only used when required and that column contains several NOPs still (especially toward the end of the schedule, not shown here). The VWR access in the 3rd column is also restricted as much as possible. The registers R1–R4 allow to store just sufficient intermediate data to achieve this highly efficient schedule. In particular, R1 stores the multiplicand on a temporary basis within the steps of the

Cycles		VWR (subwords)		R1 (sub-words)	R4 (sub-words)	SA unit			Shuffler unit (subwords)	R2 (sub-words)	R3 (sub-words)
		Rd start of cycle +/>>	Wr end of cycle	start +/end of cycle	start >>/end of cycle		Sh	Add		end of cycle	end of cycle
1		im12_a(4) >> & +				M4_12_a= GSA1(im12_a, im_12_a)	>> 3	+		M4_12_a (4)	
2		im12_b (4) >> & +			-/M4_16_a(3)	M4_12_b= GSA1(im12_b, im12_b)	>> 3	+	M4_16_a= REPACK(M4_12_a) (3)	M4_12_a (4)	M4_12_b (4)
3		im16_a (3) + & R1		-/im16_a (3)	M4_16_a>>/ M4_16_b_mxd (3)	M4_16_a= GSA1(M4_16_a, im16_a)	>> 2	-	M4_16_b_mxd= REPACK(M4_12_a, M4_12_b) (3)	M4_16_a (3)	M4_12_b (4)
4		im16_b_mxd= im16_a(1)& im16_b(2) +			M4_16_b_mxd >>/M4_16_a(3)	M4_16_b_mxd= GSA1(M4_16_b_mxd, im16_b_mxd)	>> 2	-	M4_16_a >> 5	M4_16_b_mxd (3)	M4_12_b (4)
5			M4_16_a (3)	im16_a +/- (3)	M4_16_a>>/ M4_16_b_mxd(3)	M4_16_a= GSA1(M4_16_a, im16_a)	>> 0	+	M4_16_b_mxd >> 5		M4_12_b (4)
6		im12_c (4) >> & +				M4_12_c= GSA1(im12_c, im12_c)	>> 3	+		M4_12_c (4)	M4_12_b (4)
7		im16_b_mxd + (3)	M4_16_b_mxd (3)		M4_16_b_mxd >>/M4_16_c_mxd (3)	M4_16_b_mxd = GSA1(M4_16_b_mxd, im16_b_mxd)	>> 0	+	M4_16_c_mxd= REPACK(M4_12_c, M4_12_b)(3)	M4_12_c (4)	
8		im8[x][y-1] >> (6) im8[x][y+1] + (6)	M4_16_c (3)		-/ M4_16_c_mxd (3)	M0_8_a=GSA1 (im8[x][y-1], im8[x][y+1])	>> 0	+	M4_16_c= REPACK(M4_12_c) (3)		M0_8_a (6)
9		im8[x+1][y-1 >> (6) im8[x+1][y+1 + (6)	M0_12_a (4)		-/ M4_16_c_mxd (3)	M0_8_b=GSA1 (im8[x][y-1], im8[x][y+1])	>> 0	+	M0_12_a= REPACK(M0_8_a) (4)	M0_8_b (6)	M0_8_a (6)
10		im16_c_mxd= im16_b(2)& im16_c(1) +& R1	M0_12_b (4)	-/im16_c_mxd (3)	M4_16_c_mxd >>/ - (3)	M4_16_c_mxd = GSA1(M4_16_c_mxd, im16_c_mxd)	>> 2	-	M0_12_b= REPACK(M0_8_a, M0_8_b) (4)	M0_8_b (6)	M4_16_c_mxd (3)
11		im16_c (3) + M4_16_c (3) >>			-/ M4_16_c_mxd(3)	M4_16_c = GSA1(M4_16_c, im16_c)	>> 2	-	M4_16_c_mxd >> 5	M0_8_b (6)	M4_16_c(3)
12			M4_16_c_mxd (3)	im16_c_mxd+/ - (3)	M4_16_c_mxd>>/ M4_16_c (3)	M4_16_c_mxd = GSA1(M4_16_c_mxd, im16_c_mxd)	>> 0	+	M4_16_c >> 5	M0_8_b (6)	
13		im16_c (3) +	M4_16_c (3)		M4_16_c>>/- (3)	M4_16_c = GSA1(M4_16_c, im16_c)	>> 0	+		M0_8_b (6)	
14		M0_12_a (4) >> & +	M0_12_c (4)		-/ -	M7_12_a= GSA1(M0_12_a, M0_12_a)	>> 4	-	M0_12_c= REPACK(M0_8_b) (4)	M7_12_a (4)	
15		M0_12_b(4) >> & +			-/ M7_16_a (3)	M7_12_b= GSA1(M0_12_b, M0_12_b)	>> 4	-	M7_16_a= REPACK(M7_12_a) (3)	M7_12_a (4)	M7_12_b (4)
16		M0_16_a(3) + & R1		-/M0_16_a (3)	M7_16_a>>/ M7_16_b_mxd (3)	M7_16_a= GSA1(M7_16_a, M0_16_a)	>> 3	+	M7_16_b_mxd= REPACK(M7_12_a, M7_12_b)(3)	M7_16_a (3)	M7_12_b (4)
17		M0_16_b_mxd= M0_16_a(1)& M0_16_b(2) +&R1			M7_16_b_mxd >>/M7_16_a(3	M7_16_b_mxd = GSA1(M7_16_b_mxd, M0_16_b_mxd)	>> 3	+	M7_16_a >> 2	M7_16_b_mxd (3)	M7_12_b (4)

Figure 11.46: Optimized data-path scheduling entangled with foreground data memory scheme for SoftSIMD Gauss x loop, based on [Psy10]

shift-add constant multiplications. R2 and R3 store two neighbouring words from which the shuffler will select the desired subwords for the output word. And R4 is used as temporary storage in the Shift-Add feedback loop. A small fragment of this schedule is enlarged in Figure 11.47. It contains only the 14th and 15th cycle. Here it can be seen that the optimized orchestration of the different slots allows to read the different words from the VWR, shift and add them, and reorganize them from 8- to 12-bit subwords in the shuffler to

cycles		VWR (subwords)		R1 (sub-words)	R4 (sub-words)
		Rd start of cycle +/>>	Wr end of cycle	start +/ end of cycle	start >>/end of cycle
14		M0_12_a (4) >> & +	M0_12_c (4)		-/-
15		M0_12_b (4) >> & +			-/M7_16_a (3)

SA unit			Shuffler unit (sub-words)	R2 (sub-words)	R3 (sub-words)
	Sh	Add		end of cycle	end of cycle
M7_12_a= GSA1(M0_12_a, M0_12_a)	>> 4	-	M0_12_c= REPACK(M0_8_b) (4)	M7_12_a (4)	
M7_12_b= GSA1(M0_12_b, M0_12_b)	>> 4	-	M7_16_a= REPACK(M7_12_a) (3)	M7_12_a (4)	M7_12_b (4)

Figure 11.47: Excerpt of optimized data-path scheduling for SoftSIMD Gauss x loop, based on [Psy10]

prepare them for further processing later on in the schedule. The different slots have partly different loop nest formatting but that is taking care of by the distributed loop buffer approach.

This very optimized schedule has also been introduced in the Target Compiler tools CHESS and CHECKERS [Tar08]. It has been feasible to reproduce the optimized schedule with 27 cycles for this ASIP data-path, by carefully coding the application in C with a set of user-defined intrinsics (see [Psy10]). This shows the clear potential for partly automated compiler support for our energy-efficient ASIP template.

Exploiting the background memory communication scheme which is described above in Section 11.5.1, the globally optimized schedule for the x loop is indirectly also illustrated in the program of Figure 11.46 (for N=576 bit, M=48 bit). Indeed, the VWR read and write accesses should be preceded or followed by the wide SRAM write resp. read accesses. For the y loop the schedule stays the same as the one described in Program 3 but after applying the appropriate unrolling factor (see also [Kri09]).

11.6 Energy results and discussion

As already mentioned in the previous sections, the initial energy estimation derived from the COFFEE tool was inaccurate up to now, due to the sub-optimal modeling of the new inserted FUs that implement the new intrinsics. The present section determines the accurate estimation of the energy that is

consumed in the data path of the ASIP platform. The energy consumption of the data path consists of the FU and the distributed LB energy. The distributed LB is used and it is responsible for storing the control bits for the correct execution of the instructions.

First, the consumed energy for the optimized sequential critical loop of the 2D Gauss filtering will be estimated and compared with the original code. Then, the effect of applying data parallelisation on the critical Gauss loop will be compared with the optimized sequential version of the code. Moreover, the overall energy consumption of mapping the data parallel version to the proposed architecture will take place. Finally, an estimation of the energy gain of the whole detection algorithm due to the bit true and approximate optimizations will be performed. The energy consumption will be estimated for the data path of the ASIP platform with one data cluster and two slots per data clusters, which seems to be the most attractive one.

11.6.1 Data path energy estimation for critical Gauss loop of scalar ASIP

The data path energy of the ASIP platform consists of the consumed energy by the FUs and the LB, which stores the control bits of the FUs. The background and foreground data memory operations and the instructions corresponding to address computation are outside the scope of the present section and they will be presented in Section 11.6.5 (see also [Cat09]). For the energy estimation we have considered the 40 nm TSMC technology (LP), a Vdd of 1 V, a clock frequency of 200 MHz and a data path width of 16 bits. The Table 11.5 illustrates the energy that each FU consumes. The adder and shifter are based on accurate synthesis results, starting from VHDL code. The shifter is assumed to have a limited shift factor though. When that shift factor goes up to the required range of 5 bits, the power will go up to about 40 μW. However the total power for the entire SA datapath logic should not exceed 100 μW, including the power of the pipeline register and the muxes. For the 16×16 bit multiplier a rather optimistic estimate has been made, because some results indicate a loss of up to a factor of 20 compared to a 16 bit adder.

FU	Power per activation (μW)	Energy per activation (fJ)
Multiplier	810	4050
Adder	45	225
Shifter	20	100
SA	100	500

Table 11.5: Power and energy consumption for the different FUs

Cycles	Slot 0		Slot 1	
1	ADD W	Address: (x-1,y)	ADD W	Address: (x+1,y)
2	ADD W	Address: (x,y)	L W C1 C1	Load pixel: (x,y-1)
3	ADD W	Address: (x,y-1)	L W C1 C1	Load pixel: (x+1,y)
4	ADD W	Address: (x,y-2)	L W C1 C1	Load pixel: (x,y)
5	SA1	MulRes0=(x-1)(y)+(x+1)(y)	L W C1 C1	Load pixel: (x,y-1)
6	SA3	MulRes7=MulRes0*Coef(0,0)	SA3	MulRes4=Imsub(x,y)*Coef(0,1)
7				
8				
9	SA1	ImGauss_x(x,y)=MulRes4+MulRes4	SA1	MulRes1=Imgauss_x(x,y-1)*Vector(0,1)
10	SA1	MulRes2=ImGauss_x(x,y-1)+MulRes1	L W C1 C1	Load pixel: (x,y-2)
11	ADDL W	y++	S W C1	Store pixel: ImGauss_x(x,y)
12	SA1	MulRes1=ImGauss_x(x,y-2)+MulRes2	ADD W	
13	SA1	ImGauss(x,y-1)=ImGauss(x,y-1)>>Dec	ADD W	
14			S W C1	Store pixel: ImGauss(x,y-1)
15	BRF B B F			

Figure 11.48: Software scheduling of COFFEE tool for SA FU

The FU energy is estimated by Eq. 11.9, where the E_M is the energy consumed by one activation of the multiplier and E_{SA} the energy consumed by one activation of the SA FU. In order to estimate the total energy consumed, the FU energies should be multiplied by the total number of activations.

$$E_{FU} = (E_M \times \#MUL) + (E_{SA} \times \#SA) \tag{11.9}$$

The total number of activations is equal to the number of operations performed in one iteration of the loop multiplied by the total number of iterations. The operations are derived from the scheduling of the COFFEE tool, as illustrated in Figure 11.48, excluding the operations that are responsible for the addressing of the operands and the load and store instructions. The latter are namely executed on the power-efficient DMA engine.

The energy estimation of the FUs of the architecture with one data cluster and two slots for the critical Gauss loop is depicted in Figure 11.49.

The LB energy is estimated by two bounds, one pessimistic and one optimistic. The E_{RMul} and E_{RSA} are the required energies for reading one multiplication and one SA instruction from the LB, respectively. The reading energy of one instruction is calculated by multiplying the energy consumed when reading one Flip Flop (FF) ($E_{FF} = 5\text{fJ}$) by the number of required FFs for storing the control bits of the instruction, e.g. instruction width *W*. The W_{MUL} and W_{SA} are 6 bits and 12 bits. In order to estimate the total energy consumed by the LB, the energy of reading each instruction should be multiplied by the total number of reads. The optimistic approach is calculated by Eq. 11.10. Here it is assumed that selecting the appropriate LB word to be read is not reflected in the access energy, which provides of course only a lower bound.

$$E_{LBopt} = (E_{RMul} \times \#MUL) + (E_{RSA} \times \#SA) \tag{11.10}$$

where $E_{RMul} = E_{FF} \times W_{MUL}$ and $E_{RSA} = E_{FF} \times W_{SA}$

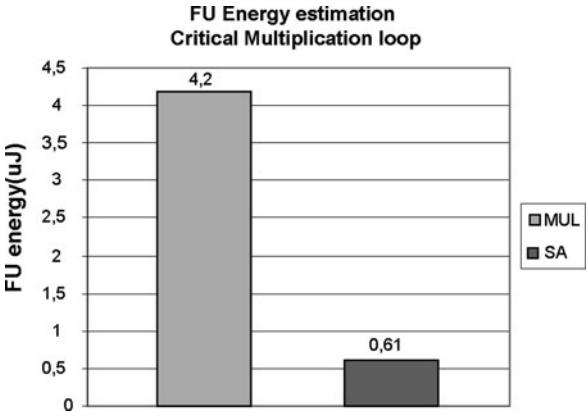


Figure 11.49: FU energy comparison of the original critical Gauss loop implemented to the multiplier FU and of the optimized critical Gauss loop implemented to the SA FU

Figure 11.50: LB structure

In the most pessimistic energy estimation of the LB it is assumed that the read energy is proportional to the depth of the LB, as shown in Figure 11.50. The consumed energy is calculated from the Eq. 11.11. The real energy consumption is somewhere between these bounds. For the further experiments the upper bound is selected to ensure that the overall gains are not overestimated later on.

$$E_{LBpess} = (E_{LBOpt} \times LB_{depth}) \tag{11.11}$$

The energy estimation of the LB for the architecture with one data cluster and two slots for the critical Gauss loop is depicted in Figure 11.51

11.6.2 Data path energy estimation for critical Gauss loops of SoftSIMD ASIP

The same assumptions as in Section 11.6.1 are made for the data path energy of the ASIP platform, except that the data path width is now 48 bits. Table 11.6 illustrates the energy that each FU consumes. The shuffler

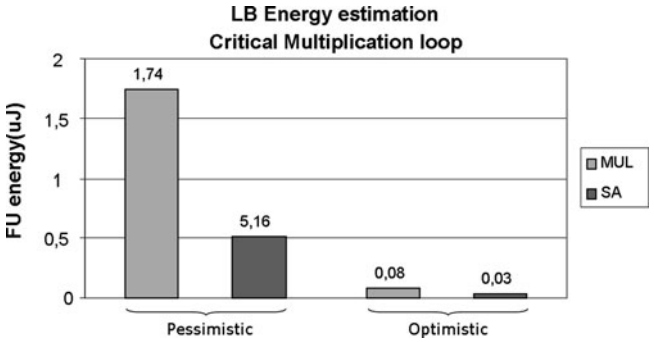


Figure 11.51: LB energy comparison for pessimistic and optimistic approach of the original critical Gauss loop implemented to the multiplier FU and the optimized critical Gauss loop implemented to the SA FU

FU	Power per activation (µW)	Energy per activation (fJ)
Shuffler	360	1,800
SA	300	1,500

Table 11.6: Power and energy consumption for the different FUs

is based on a very worst-case older VHDL synthesis experiment [Rag07c]. It should be possible to improve this energy estimation significantly based on module generation with a more optimized routing algorithm. In addition, the SA power is overestimated as it is assuming a factor of 3 more power for the 48-bit data path compared to the initial one for the 16-bit data path.

The FU energy is estimated by Eq. 11.12, where the E_{FUx} is the energy consumed by the activation of the FU due to the instructions of Soft-SIMD on x and the E_{FUy} due to the instructions of Soft-SIMD on y. In order to estimate the total energy consumed, the above energies should be multiplied by the total number of activations, respectively. The operations performed in one iteration of the loop are again derived from the scheduling of the COFFEE tool. But the operations that are responsible for the addressing of the operands and the load and store instructions have again been excluded because these are executed on a separate power-efficient AGU unit (see, e.g. options described in [Tal08]).

$$E_{FU} = (E_{FUx} + E_{FUy}) \tag{11.12}$$

where $E_{FUx} = (E_{SA} \times \#SA_x) + (E_S \times \#S_x)$ and $E_{FUy} = (E_{SA} \times \#SA_y)$ $+(E_S \times \#S_y)$.

The energy estimation of the FUs of the architecture with one data cluster and two slots for the critical Gauss loop is depicted in Figure 11.52.

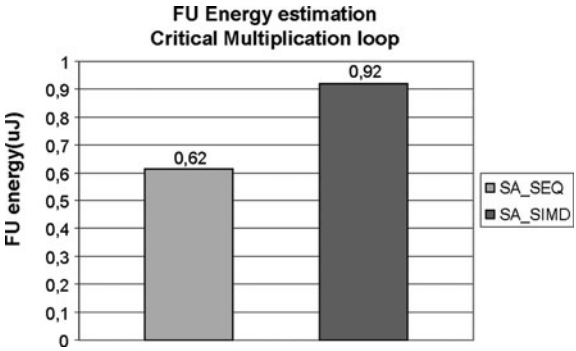


Figure 11.52: FU energy comparison of sequential critical Gauss loop implemented to the SA FU and parallel one using Soft-SIMD implemented on the Shift-Add-Shuffler FU

The LB energy is estimated by the pessimistic approach. The E_{RSAS} is the required reading energy for one instruction from the LB. The reading energy is calculated by multiplying the energy consumed when reading one Flip Flop (FF) ($E_{FF} = 5\text{fJ}$) by the number of required FFs for storing the control bits for each instruction, e.g. instruction width *W*. The W_{SA} and W_S are 6 bits and 4 bits for the shuffler. In order to estimate the total energy consumed by the LB, the consumed energy of reading each instruction should be multiplied by the total number of reads. It should be stressed that while unrolling the operations the LB reads are not increasing, due to the fact that the same instruction, i.e. the same control bits, is applied to different data. Applying again the pessimistic approach results in Eq. 11.13.

$$E_{LB} = (E_{LBx} + E_{LBy}) \tag{11.13}$$

where $E_{LBx} = E_{LBpessx}$,$E_{LBy} = E_{LBpessy}$,$E_{LBpessx} = E_{FF} \times (W_{SA} + W_{SF}) \times LB_{depth} \times \#SAS_x$ and $E_{LBpessy} = E_{FF} \times (W_{SA} + W_{SF}) \times LB_{depth} \times \#SAS_y$.

The energy estimation of the LB the architecture with one data cluster and two slots for the critical Gauss loop is depicted in Figure 11.53.

11.6.3 Data path energy estimation for overall Detection algorithm

The estimation for the data path energy consumption of the detection algorithm is performed for the mapping of the detection algorithm to the scalar ASIP platform with one data cluster and two slots per data cluster. The data

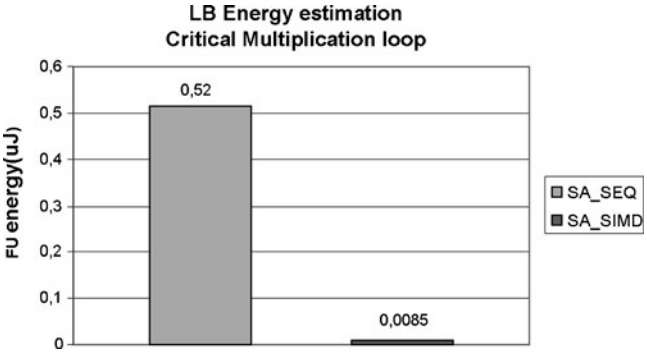


Figure 11.53: LB energy comparison of sequential critical Gauss loop implemented to SA FU and parallel critical Gauss loop using Soft-SIMD implemented on Shift-Add- Shuffler FU

FU	Power per activation (μW)	Energy per activation (fJ)
Multiplier	810	4,050
Adder	45	225
Shifter	20	100
SA	100	500
GSAS	180	900

Table 11.7: Power and energy consumption for the different FUs

path energy of the ASIP platform consists of the consumed energy of the FUs and the LB, which stores the control bits of the FUs. A similar methodology as the one that has been applied for the energy estimation of the critical Gauss loop, can be also applied for the data path energy estimation of the detection algorithm. Assuming 40 nm TSMC technology, clock frequency 200 MHz and data path width 16 bits, the Table 11.7 illustrates the energy that each FU consumes.

The FU energy can be estimated by Eq. 11.9 taking into account the whole code of the detection algorithm. The consumed energy is approximated by the most dominant part of the code, which is the loops that iterate through the whole image. This is illustrated in for the main Gauss loop in Figure 11.54.

The energy estimation of the FUs of the architecture with one data cluster and two slots for detection algorithm is depicted in Figure 11.55.

The LB energy is estimated again by the pessimistic approach where it is assumed that the read energy is proportional to the depth of the LB and

Cycles	Slot 0	Slot 1
1	ADD W	ADD W
2	L W C1 C1	L W C1 C1
3	ADD W	ADD W
4	L W C1 C1	L W C1 C1
5	MPYL W	ADD W
6	MPYL W	L W C1 C1
7	MPYL W	ADD W
8	MPYL W	L W C1 C1
9	ADD W	SHL W
10	MPYL W	SHL W
11	L W C1 C1	ADD W
12	MPYL W	SHL W
13	L W C1 C1	ADD W
14	ADDL W	SHL W
15	MPYL W	L W C1 C1
16	ADDL W	SHL W
17	MPYL W	ADDL W
18	MPYL W	SHL W
19	ADDL W	SHL W
20	ADDL W	SHL W
21	ADDL W	SHL W
22	ADDL W	ADDL W
23	ADDL W	ADDL W
24	ADD W	SHR W
25	BRF B B F	S W C1

Cycles	Slot 0	Slot 1
1	ADD W	ADD W
2	ADD W	L W C1 C1
3	ADD W	L W C1 C1
4	ADD W	L W C1 C1
5	GSAS1	L W C1 C1
6	GSAS3	GSAS3
7		
8		
9	GSAS1	L W C1 C1
10	GSAS1	S W C1
11	GSAS1	ADDL W
12	GSAS1	ADD W
13	ADD W	S W C1
14	BRF B B F	

Figure 11.54: Software scheduling of Gauss loop in MUL FU and GSAS FU

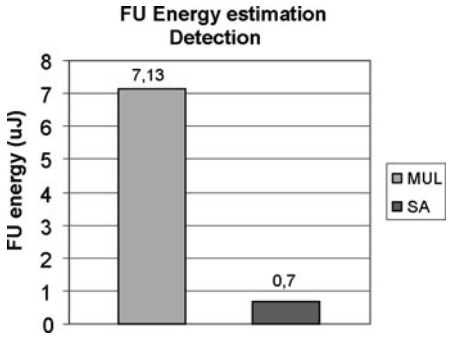


Figure 11.55: FU energy comparison of the original detection algorithm implemented to multiplier FU and transformed detection algorithm implemented on SA FU

it can be calculated by Eq. 11.11. The energy estimation of the LB for the architecture with one data cluster and two slots for the detection algorithm is depicted in Figure 11.56.

11.6.4 Energy modeling for SRAM and VWR contribution

The SRAM energy model is derived from the ultra-low power SRAM research at the K.U.Leuven-MICAS group. Both the active and the leakage energy for small embedded SRAMs up to 256kbt, have been heavily optimized in their work (see e.g. [Cos08, Gee09]). So that is the ideal starting point for our

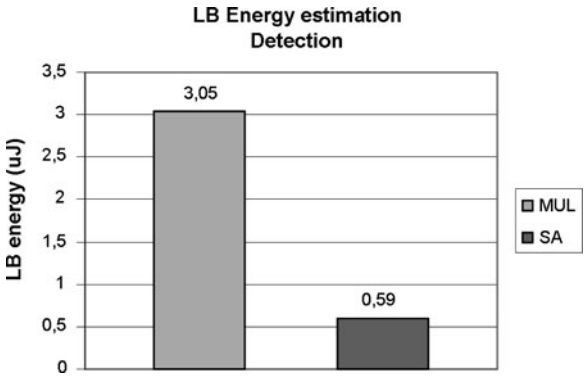


Figure 11.56: LB energy comparison of the original detection algorithm implemented on multiplier FU and transformed detection algorithm implemented on SA FU

L1 SRAM scratchpad memory. Details of this estimate will not be provided here, see [Cat09]. The result is an SRAM circuit with a quite low dynamic energy operating at 250 MHz and 0.6 V Vdd in 90 nm CMOS. When scaled to 40 nm CMOS we expect about 4.3 fJ/bit/read access and 7.4 fJ/bit/write access. For the 32–48 bit interface (needed in the version without VWR), we obtain in a similar fashion 13.5 fJ/bit/read access and 22.6 fJ/bit/write access. Obviously, the absolute accuracy of these data should not be seen as better than about 20–30%, but the relative accuracy can be expected to be much better. And for our purposes we mainly need the relative comparison.

These calculations have also made clear that for all practical purposes, in our embedded low-power context (with low temperatures and non-aggressive clock cycles) we can safely ignore leakage in the 90 nm node.

The VWR energy model is based on VHDL synthesis and comparing it to an adder of the same bit-width. A 576-bit wide VWR with 1-port cells but two external ports of 48 bit (to the data-path) would consume a pure dynamic energy of about 1,215 fJ/576-bit memory transfer and 108 fJ/48-bit data-path transfer, when operating at 200 MHz in a 40 nm TSMC CMOS node. It is assumed here that the mux selector from 576 bit lines to 48-bit wide words is implemented in a very effective style including river routing where feasible. But we have to incorporate also clock power, which have been assumed to increase the result with a factor 2 when the layout is organized in a quite regular fashion. That amounts to 2,430 fJ/576-bit memory transfer and 216 fJ/48-bit data-path transfer. Compared to the SRAM read data above that is already relatively high.

The register-file (RF) energy model for the non-SIMD architectures is also based on VHDL synthesis but now using absolute numbers and a conservative scaling factor of 4 form 90 to 40 nm. For the multiplier-based ASIP we require a 32-bit wide regfile with 32 words and 6 ports. The energy is then about 7,500 fJ/32-bit data-path access, when operating at 200 MHz in a 40 nm TSMC CMOS node. For the ShAdd based ASIP, we require a 16-bit wide regfile with 16 words and 6 ports. The energy is then about 2,470 fJ/16-bit data-path access.

11.6.5 Memory energy contributions

We will first study the variable multiplier based ASIP architecture. In that case we have 5,528,952 accesses to the standard-cell synthesis-based 32w*32b*6port regfile, which at 7,500 fJ/access leads to a total of about $41.47\,\mu$J. And the number of accesses to the 32-bit SRAM is then 311,040 reads and 103,680 writes so the total energy is then about $240\,$fJ, which is quite low especially due to the very low-power SRAM used here. That regfile energy is clearly highly dominant compared to the 4.2 and $1.74\,\mu$J spent in the data-path operators respectively the loop buffers. If we would have used conventional instruction caches, that contribution would have also been quite dominant in the global power pie though.

Next we will analyze the shift-add constant multiplier based ASIP architecture. Now we have $12{,}888 + 3{,}673{,}080 = 3{,}685{,}968$ accesses to the standard-cell synthesis-based 16w*16b*6port regfile, which at 2,470 fJ/access leads to a total of about $9.1\,\mu$J. And the number of accesses to the 16-bit SRAM is then also 311,040 reads and 103,680 writes so the total energy is then estimated to be about $240\,$fJ just as for the multiplier ASIP. Again the regfile energy is very dominant compared to the 0.614 and $0.516\,\mu$J spent in the data-path operators respectively the loop buffers. Also here a more conventional instruction cache solution would have been quite dominant in the global power pie.

Finally we will study the soft-SIMD architecture. In our bioimaging case study the requirements for the most critical Gauss filter loop in the detection module, requires per detection execution about $\mathrm{X} = 4950$ read accesses and $\mathrm{Y} = 5670$ write accesses of 576 bit each to the SRAM. So the total detection energy amounts to $X/10^3 * 7.481 + Y/10^3 * 13.325 = 112.6\,$nJ per image frame. For the VWR the access rate is higher due to the additional 48-bit data-path interaction. So on top of the 576-bit reads/writes (corresponding to the SRAM access), we also need X2=495,900 read accesses and Y2=204,060 write accesses of 48 bit. This results in a rather worst-case energy per detection execution of about $(X + Y)/10^3 * 2.43 + (X2 + Y2)/10^3 * 0.216 = 25.8{+}151.2 = 177\,$nJ per image frame. So the total energy in the back- plus foreground data memory organisation becomes about $290\,$nJ per image frame.

For the optimized schedule of Figure 11.46, which requires a few additional muxes in the Soft-SIMD data-path, we arrive even at a VWR contribution of $107nJ$ because then X2=25,976 and Y2=92,754, leading to a total of only $220nJ$ in the data memory organisation per image frame!

11.6.6 Global performance and energy results for options

The set of the applied systematic methodologies the bioimaging application, such as the advanced fixed point refinement of the signals in the application, the efficient conversion of the constant multiplications to sequences of shift and addition operations, and finally the effective application of the Soft-SIMD approach lead to a heavily optimized ASIP architecture with low-cost ultra low-energy consumption while it still meets all the performance requirements. The performance gain for the critical Gauss filtering loop of the bioimaging application is illustrated in Figure 11.57.

Observing the results of Figure 11.57, the obtained performance gain due to quantization techniques that lead to the fixed point refinement of the critical Gauss loop is equal to 5.14. The application of bit true code transformations in order to reduce the number of the constant multiplications and furthermore to reduce the number of shift and addition operations, has a performance gain of 1.76. The conversion from constant multiplications to shift and addition operations does not have a great impact of performance, while it will enable the application of Soft-SIMD and thereby a significant reduction of the energy consumption, especially in the instruction and data memory hierarchies. Finally, the enabling of data parallelization due to Soft-SIMD has as result a further gain factor of 2.42. The total performance gain

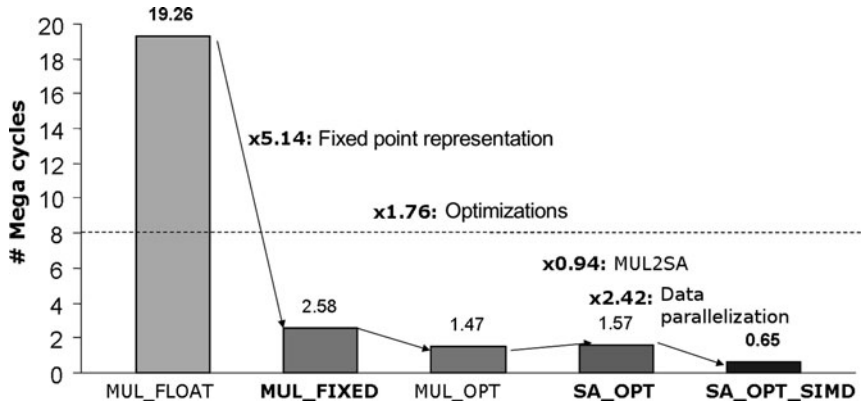


Figure 11.57: Performance of the critical Gauss loop for the different implementations

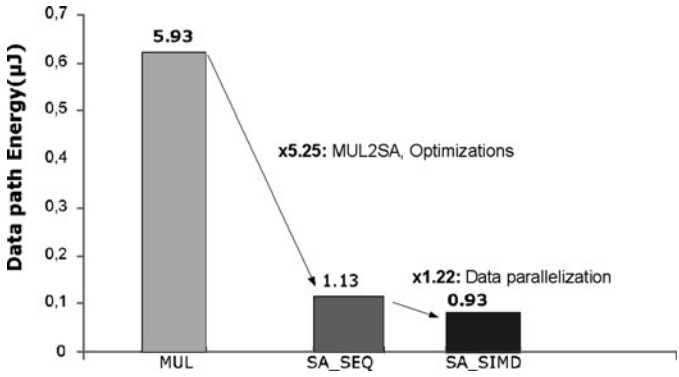


Figure 11.58: Data path energy consumption of the critical multiplication loop for the different implementations

for the critical constant multiplication loop is actually equal to 29.63. Energy gains are expected by fully avoiding the multiplication operations. The data path energy estimation of the processor with one data cluster and two slots per data cluster is depicted in Figure 11.58.

Observing the results of Figure 11.58, the obtained energy gain of the data path of the processor due to the conversion from fixed-point constant multiplications to shift and addition operations, while applying code transformations in order to reduce the number of the constant multiplications and to reduce the number of shift and addition operations, has resulted in a gain of 5.25. Finally, the enabling of data parallelization due to Soft-SIMD has resulted in an additional gain factor of 1.22. The latter is however an underestimate because several pessimistic estimates have been included here, and the main expected gain is situated in the data memory organization (see Section 11.6.5 and [Cat09]).

When we use the estimates of Section 11.6.5 combined with the results for the Shift-Add-Shuffler data-path leads to the total energy that is depicted in Figure 11.59 The obtained energy gain of the processor due to the conversion from fixed-point constant multiplications to shift and addition operations, while applying code transformations in order to reduce the number of the constant multiplications and to reduce the number of shift and addition operations, has resulted in a gain equal to 4.45. Finally, the enabling of data parallelization due to Soft-SIMD has as result an additional gain factor of 9.31. The data path energy is clearly dominant now compared to the energy consumption of the data background memory and the VWR foreground memory for the Soft-SIMD. Note that also the instruction memory overhead (the loop buffer mainly) represents only 0.08% of the total energy due to the use of the distributed loop buffer approach of Chapter 5.

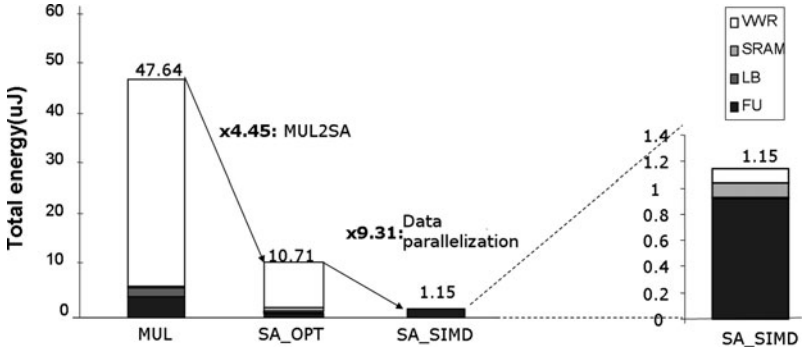


Figure 11.59: Total energy estimation of the Gauss loop for the implementation with multiplier, SA and Soft-SIMD

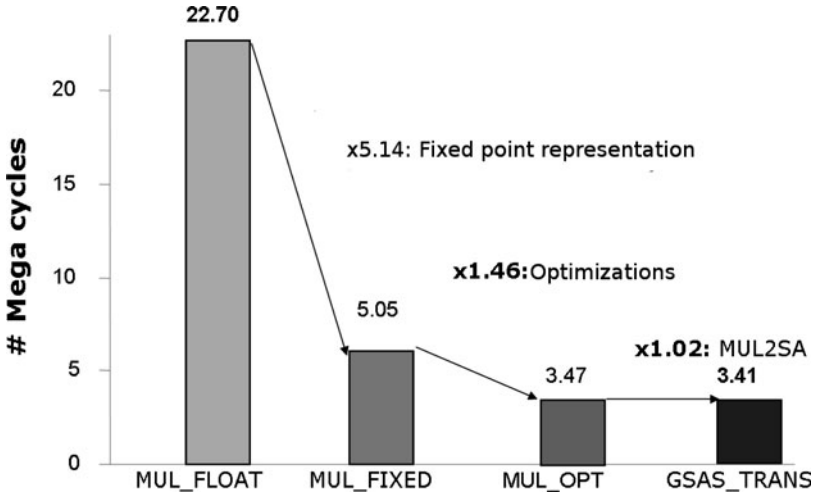


Figure 11.60: Performance of full detection algorithm for the different implementations

For the whole detection algorithm, more generic Function Units (FUs) that implement the sequences of shift and add operations have been introduced in order to be able to integrate it. This has enabled to also convert costly operations like division, square root and trigonometric functions. A thorough exploration of the most promising ASIP architectures, which can integrate the biotechnology application, has been performed. The overall performance gain for the detection algorithm of the biotechnology application is illustrated in Figure 11.60.

Observing the results of Figure 11.60, the obtained performance gain due to quantization techniques is equal to 4.5. The conversion from constant

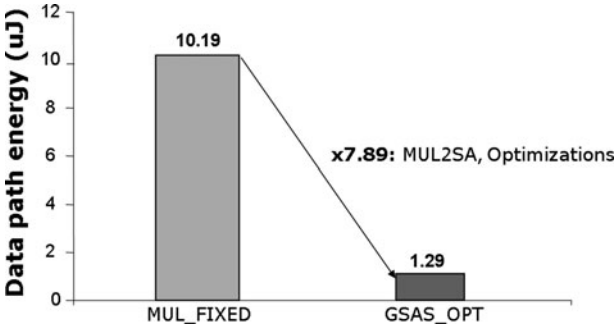


Figure 11.61: Data path energy estimation of the full detection algorithm for the different implementations

multiplications to shift and addition operations and the mapping of the detection algorithm to GSAS FU, while applying bit true code transformations, such as multiple precision multiplier and Look Up Tables, has a performance gain equal to 1.48. Finally, the total performance gain is equal to 6.66. Based on the experiments of the critical multiplication loop, we expect that the enabling of data parallelization due to Soft-SIMD will lead to even better performance gains. The detection algorithm that uses the multiplier FU has 1/6 of the total operations equal to multiplication operations. By reducing the required multiplication operations to a small number that is implemented in a 4-bit multiplier further energy gains are expected. The data path energy estimation of the processor with one data cluster and two slots per data cluster is depicted in Figure 11.61.

Observing the results of Figure 11.61, the obtained energy gain of the data path of the processor due to the conversion from constant multiplications to shift and addition operations, while applying code transformations in order to reduce the number of the multiplications is equal to 7.89. Finally, based on the experiments of the critical constant multiplication loop the enabling of data parallelization due to Soft-SIMD will lead to further energy savings.

The energy efficiency of the dominant part of detection algorithm is about 1217 MOPS/mW for an estimated 40 nm TSMC implementation, regarding the Soft-SIMD data-path including the instruction memory overhead. A first estimate of the energy in the SRAM and VWR leads to the conclusion that those contributions are smaller, namely only 13% extra overhead. The latter still achieves real-time with a clock frequency of about 100 MHz or with a 50% duty-cycles clock frequency of 200 MHz. This clearly enables a battery-operated mobile system with long lifetime for the on-line animal tracking system.

However, we have then not yet counted the energy consumed in the AGU data-path and corresponding loop buffers. Given that the access patterns are

quite regular, and the total number of addresses to be supplied is really low, it can be expected though that this addressing overhead is so low it can be mostly ignored (compared to the actual reads and writes to the SRAM). Also the control overhead should be negligible on condition that we use the distributed loop buffer approach of Chapter 6. Detailed layout experiments on a full netlist will have to confirm this though.

We can conclude that the data memory organisation in the soft-SIMD version of the bioimaging ASIP has become a minor cost. This is in sharp contrast with the initial sequential multiplication option for a more conventional VLIW processor architecture, where the data and instruction memory organisation heavily dominate the global power pie [Lam04, Hul08]. Actually, an initial estimate has shown that **compared to a heavily optimized standard-cell based ASIC implementation, we loose less than a factor 2 in energy efficiency!**

11.7 Conclusions and key messages of chapter

This chapter has described the application of the main techniques proposed in this book to a realistic application benchmark, namely a bioimaging detection and tracking algorithm for on-line animal monitoring. First the application itself has been discussed, including the fixed-point word-length quantisation of the variables. Then we have discussed our efficient proposal to realize the many constant multiplications in the application code. Next, we have described the different architecture options that are energy and performance optimized for the ASIP template that we have explored: the scalar option, the data-path and background/foreground memory organisation for the data-parallel option. template, with emphasis on the data-path.

Finally, we have provided the overall energy results and a discussion of the global ASIP exploration. Thanks to the effective and holistic application of the techniques introduced in this book, we have obtained an overall energy efficiency of over 1,000 MOPS/mW for a standard cell based 40 nm TSMC CMOS library and a low-power SRAM macro. That figure includes the data and instruction memory organisation of the ASIP platform. So we believe that this result is about 20 times better than state-of-the-art low power DSP platforms, and at least 10 times better than other proposed instruction-set programmable ASIP platforms for a similar wide scope of the application domain target. More information is also available in [Kri09].

CHAPTER 12

Conclusions

This book has presented various aspects of low-energy domain-specific instruction-set processor architecture exploration for embedded systems.

12.1 Related work overview

Chapter 2 presents an extensive overview of related work with a focus on low-power/low-energy platform architecture exploration and design. It is clearly evident that the design space for embedded systems is extremely large. For instance the number of options available for the design of a processor core and the associated instruction memory organization grows exponentially. Furthermore, a complete embedded platform consists of multiple components that can not be studied in isolation, as modifications to one component will influence various other components and will also add to the overall cost metrics in non-obvious ways. Reliance on a template architecture and a corresponding consistent architecture exploration framework becomes essential. To be able to perform an efficient exploration and compare different processor styles, the exploration should be performed early on in the design cycle. Therefore a trade-off has to be made between the implementation effort that is required to model an architecture, the evaluation speed and the accuracy.

F. Catthoor et al., *Ultra-Low Energy Domain-Specific Instruction-Set Processors*,
DOI 10.1007/978-90-481-9528-2_12,

12.2 Ultra low energy architecture exploration

Chapter 3 first describes our case studies on energy/power and performance analysis on a realistic embedded platform for a video encoding application. Through these studies we have been able to derive the requirements of a template architecture for low-energy high-performance embedded processors.

The platform study provide us with important insights regarding the influence of choices for the main components on each other and also to the overall cost metrics. Specifically this has lead us to conclude that the data memory hierarchy, the instruction memory organisation and the processor (foreground memory plus data-path units), consume about equally important parts of the platform power pie. The absolute cost of the datapath (only the Functional Units) is quite low, but, as the processor style significantly influences the cost of the foreground memory and the instruction memory organisation, the importance of the processor style (and hence the datapath) is still quite considerable.

The processor design style case study shows that improvements for energy efficiency and/or performance over state-of-the-art RISC or VLIW processors can be achieved. The results indicate that a RISC processor can perform a task with the small number of operations. However even for higher clock frequencies, the RISC style does not provide the performance needed for real-time multimedia applications. A centralized VLIW processor exploits its extra resources to increase performance, but the energy cost of a centralized registerfile (foreground memory) will be prohibitive to integrate this type of processor into platform with stringent energy constraints. A clustered VLIW can partly bring the energy consumption down, initially at the cost of a marginal compromise in performance. However after a certain point, the performance deteriorates significantly due to inter-cluster communication and the inability of the current compilers to utilize all the clusters. A coarse grain reconfigurable array (CGRA) with tightly coupled VLIW can boost the performance of applications that are regular enough to keep the vast amount of resources busy. However, also this comes at the price of overheads in the number of operation and interconnect and thus an increase in energy cost for the same task.

Another major focus of this chapter is the design/derivation of a scalable high-level architecture design template called FEENECS. It is targeted at low energy, high performance embedded systems for data dominated and streaming applications. This template has been derived based on a thorough analysis and reasoning, incorporating the constraints from application, architecture and deep submicron implementation, especially the incorporation of the dominance of interconnect in the future scaled technologies.

One of the major foundations of this work is the building of a consistent framework for performing architecture exploration. The COFFEE framework presented in Chapter 4 is capable of modeling and performing exploration over a large range of architectural parameters for a large range of benchmarks and industrial applications. This framework is capable of compiling, simulating and deriving key performance metrics like area, energy, power consumption, performance measured in cycles, Instruction Level Parallelism, etc. While this is originally based on the Trimaran 2.0 compilation framework, we have over the years extended this framework to incorporate various additional features as explained in Chapter 4. The accuracy of the energy estimation has been validated by comparing the figures from the framework to a detailed gate level simulation of an in-house embedded processor. Several case studies are shown in Chapter 4 to illustrate and explain various counter-intuitive trends that occur during architecture exploration. Essentially, the proposed framework provides all the necessary low power architecture features to optimize processors for low-power high-performance embedded systems.

12.3 Main energy-efficient platform components

The next chapters present the individual key novel architectural extensions that improve the energy efficiency over the respective state of the art.

Distributed loop buffer and multi-threading: A major contribution of this book is the concept of distributed or clustered loop buffering. Chapter 5 presents the basic concept of distributed loop buffering along with different schemes for the local controller. It has also been indicated that local controllers with activation trace and index translation offer more freedom in the architecture. In comparison to other centralized schemes, distributed loop buffer schemes consume about 63% lower energy consumption without compromising performance. Since this clustering/distribution scheme can be applied to both data (foreground memories) and instructions, a scheme has also been presented to remove any incompatibility between the two different types of clusters without compromising significantly on any of the cost metrics. Although the notion of clustering/distributing is applied here to loop buffers, in principle the same concept can be applied to any other instruction memory components e.g. the level 1 instruction caches.

Chapter 6 has presented the concept how the distributed loop buffer scheme can be extended to also support the execution of multiple incompatible loops in parallel. Such an architecture has been shown to provide efficient local communication while effectively enabling to run multiple threads in parallel. Through simulation studies it has also been shown to be highly energy

efficient compared to state-of-the-art simultaneous multi-threading (SMT) techniques, as well as other techniques like loop merging.In addition,a proposal for building a compiler technique supporting this architecture extension has also been provided.

Irregular data access support on scratchpad memories: Many applications in our target domain contain at least a few irregular indexes and dynamic data accesses. Normally, these are not supported on the platform architecture and its compiler. In Chapter 7 we have described how we can extend the framework to handle these extended data access schemes on the background data memory organisation based on compiler-driven scratchpads.

The proposed techniques and algorithms explore the appropriate portions of the arrays or irregular data structures to be brought into the scratch-pad at different time instants, such that the total energy consumption is minimized. We have also demonstrated that our technique works well on real-life applications using cache models.

Very Wide Register – foreground memory alternative: Another core contribution of this book is the asymmetrical register file organisation, called very wide register (VWR), in Chapter 8. The proposed VWR has been shown to be inspired from both application characteristics as well as deep submicron technology aspects. This VWR together with its interface to the wide layer-1 SRAM scratchpad, achieves a significantly higher energy efficiency than conventional organizations and forms a layout-friendly solution for applications with significant spatial locality. It has been demonstrated through a standard-cell implementation that the proposed architecture even then gains substantially over a multi-ported register file.

Software SIMD approach: Exploiting word-width information during application mapping is key to reduce the energy consumption or to improve the performance of processor-based embedded systems. In Chapter 9, we have presented the use of word-width aware energy models to improve energy estimation sensitivity to word-width variation in instruction-set processor simulation. We have also systematically described how to exploit this information during various steps of the mapping process, namely during assignment, scheduling, ISA selection and parallelization. For each part, the concept of the optimization is detailed and the expected gains are evaluated. Next, we have presented a more detailed description of how to implement word-width aware parallelization, also called Software SIMD.

In addition, in Chapter 10, we have described how we can apply strength reduction for the main operations that occur in our target domain. One major class are the constant multiplications. We have presented a systematic overview of the complete conversion space. A context-aware cost-driven search over this space is proposed.

Bioimaging demonstrator: To substantiate our holistic claims of combining high performance with overall energy-efficiency, in Chapter 11, we have combined the main techniques proposed in this book. They are applied to a realistic application benchmark, namely a bioimaging detection and tracking algorithm for on-line animal monitoring. Most of the components and contributions presented in this book have been applied and illustrated in this realistic demonstrator. The end result is a domain-specific processor instance which can reach about 1,000 MOPS/mW in a 40 nm TSMC CMOS technology with standard-cells for the data-path and an SRAM module generator for the memories. Both the data and instruction memory organisation are included in that figure-of-merit.

Bibliography

[Abb07] A.Abbo, R.Kleihorst, V.Choudhary L.Sevat, P.Wielage, S.Mouy, and M.Heijligers. Xetal-ii: A 107 gops, 600mw massively-parallel processor for video scene analysis. *Proc. of ISSCC*, 2007.

[Abs05] J.Absar and F.Catthoor. Compiler-based approach for exploiting scratch-pad in presence of irregular array access. *Design Automation and Test in Europe (DATE)*, pages 1162–1167, March 2005.

[Abs06] J.Absar and F.Catthoor. Reuse analysis of indirectly indexed arrays. *ACM Trans. on Design Automation of Electronics Systems (TODAES)*, 11(2):282–305, 2006.

[Abs07] J.Absar. Locality Optimization in a Compiler for Embedded Systems. PhD thesis, KULeuven, July 2007.

[Abs08] J.Absar, P.Raghavan, A.Lambrechts, M.Li, M.Jayapala, and F.Catthoor. Locality optimizations in a compiler for wireless applications. *Design Automation of Embedded Systems (DAEM)*, April 2008.

[ACE08] ACE Inc., http://www.ace.nl/compiler/cosy.html. CoSy Compiler Development System, 2008.

[Adi00] S.Aditya, S.Mahlke, and B.Rau. Code size minimization and retargetable assembly for custom epic and vliw instruction formats. *ACM Trans. Des. Autom. Electron. Syst. (TODAES)*, 5(4):752–773, 2000.

[Aga89] A.Agarwal, J.Hennessy, and M.Horowitz. An analytical cache model. *ACM Trans. on Computer Systems*, 7(2):184–215, 1989.

[All81] F.E.Allen, J.Cocke, and K.Kennedy. Reduction of Operator Strength, In Program Flow Analysis:Theory and Applications. Prentice Hall, NJ, USA, 1981.

[Amr00] B.Amrutur and M.Horowitz. Speed and power scaling of SRAM's. *IEEE Journal of Solid-State Circuits*, Vol.35, February 2000.

[And00] T.Anderson and S.Agarwala, Effective hardware-based two-way loop cache for high performance low power processors. *Proc. of Intnl. Conf. on Computer Design (ICCD)*, September 2000.

[Ann90] F.Annexstein, M.Baumslag, M.Herbordt, B.Obrenic, A.Rosenberg, and C.Weems, Achieving multigauge behavior in bit-serial simd architectures via emulation. *3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 186–195, 1990.

[ARM] ARM, http://www.arm.com/products/physicalip/memory.html. Artisan Memory Generator.

[ARM09a] ARM, http://www.arm.com/products/CPUs/families/ARM7Family.html. ARM 7 Family, January 2009.

[ARM09b] ARM, http://www.arm.com/products/CPUs/families/ARM9Family.html. ARM 9 Family, January 2009.

[Asa98] K.Asanović. Vector Microprocessors. PhD thesis, University of California Berkeley, 1998.

[Asc03] G.Ascia, V.Catania, M.Palesi, and D.Patti. Epic-explorer: A parameterized vliw-based platform framework for design space exploration. *Proc. of ESTIMedia*, pages 3–4, 2003.

[Ask99] S.Askar and M.Ciesielski. Analytical approach to custom datapath design. *ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 98–101, 1999.

[AT90] AT & T, http://www.att.com. AT&T DSP1600 Microprocessor, 1990.

[ATI09] ATI, http://ati.amd.com/products/mobilityradeonhd3600/index.html. ATI Radeon 3600 Series Processor, January 2009.

[Ati07] D.Atienza, P.Del Valle, G.Paci, F.Polett, L.Benini, G.De Micheli, J.M.Mendias, and R.Hermida. Hw-sw emulation framework for temperature-aware design in mpsocs. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):1–26, 2007.

[Aus02] T.Austin, E.Larson, and D.Ernst. Simplescalar: an infrastructure for computer system modeling. *IEE Computer Magazine*, 35(2):59–67, 2002.

[Avi02] O.Avissar and R.Barua, An optimal memory allocation scheme for scratch-pad based embedded systems. *ACM Trans. on Embedded Computing Systems*, pages 6–26, November 2002.

[Baj97] R.Bajwa, M.Hiraki, H.Kojima, D.J. Gorny, K.Nitta, A.Shridhar, K.Seki, and K.Sasaki. Instruction buffering to reduce power in processors for signal processing. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 5(4):417–424, December 1997.

[Bal98] S.Balakrishnan and S.Nandy. Arbitrary precision arithmetic-simd style. *Eleventh Intnl. Conf. onVLSI Design, 1998*, pages 128–132, 1998.

[Bal04] A.Balakrishnan. An experimental study of the accuracy of multiple power estimation models. Master's thesis, University of Tennesse, Knoxville, August 2004.

[Ban02] R.Banakar, S.Steinke, B.Lee, M.Balakrishnan, and P.Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. *Proc. of the 10th Intnl. Symposium on Hardware/-software Codesign, CODES'02*, pages 73–78, May 2002.

[Ban04] N.Bansal, S.Gupta, N.Dutt, A.Nicolau, and R.Gupta. Network topology exploration of mesh-based coarse-grain reconfigurable architectures. *Proc. of Design Automation and Test in Europe (DATE)*, 2004.

[Bar05a] F.Barat. CRISP: A Scalable VLIW Processor for Multimedia Applications. PhD thesis, KULeuven, ESAT/ELECTA, 2005.

[Bar05b] M.Baron. Cortex a8:high speed, low power. Microprocessor Report, October 2005.

[Ben01] L.Benini, D.Bruni, M.Chinosi, C.Silvano, V.Zaccaria, and R.Zafalon. A power modeling and estimation framework for vliw-based embedded systems. *PATMOS Intnl. Symposium*, 2001.

[Ben02] L.Benini, D.Bruni, M.Chinosi, C.Silvano, and V.Zaccaria. A power modeling and estimation framework for vliw-based embedded system. *ST Journal of System Research*, 3(1):110–118, April 2002.

[Ben05a] L.Benini, D.Bertozzi, A.Bogliolo, F.Menchelli, and M.Oliviri. Mparm: Exploring the mulit-processor soc design space with systemc. *Journal of VLSI Signal Processing*, volume 41(2), pages 169–182, 2005.

[Ben05b] L.Benini, D.Bertozzi, A.Bogliolo, F.Menichelli, and M.Olivieri. Mparm: Exploring the multi-processor soc design space with systemc. *J. VLSI Signal Process. Syst.*, 41(2):169–182, 2005.

[Ber86] R.Bernstein. Multiplication by integer constants. *Softw. Pract. Exper.*, 16(7):641–652, 1986.

[Ber06] Berkeley Design Technology, Inc. (BDTi), http://www.dspdesignline.com/showArticle.jhtml?articleID=192501356. Emulating SIMD in Software, 2006.

[Bes03] A.Besdéz, R.Ferenc, T.Gyimttthy, A.Dolenc, and K.Karsisto. Survey of code-size reduction methods. *ACM Computing Surveys (CSUR)*, 35(3):223–267, September 2003.

[Bou08] B.Bougard, B.De Sutter, S.Rabou, D.Novo Bruna, O.Allam, S.Dupont, and L.Van der Perre. A coarse-grained array based baseband processor for 100mbps+ software defined radio. *Proc. of DATE*, 2008.

[Bou98] P.Boulet, A.Darte, G.Silber, and F.Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing*, 24(3–4):421–444, 1998.

[Bri94] P.Briggs and T.Harvey. Multiplication by integer constants. Technical report, Rice University, 1994.

[Bro00a] E.Brockmeyer, C.Ghez, W.Baetens, and F.Catthoor. Unified low-power design flow for data-dominated multi-media and telecom applications. Kluwer Acad Publ. Boston, 2000.

[Bro00b] D.Brooks, V.Tiwari, and M.Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. *Proc. of the 27th Intnl. Symposium on Computer Architecture (ISCA)*, pages 83–94, June 2000.

[Bun93] J.Bunda. Instruction-processing optimization technique for vlsi microprocessors. Phd dessertation, University of Texas at Austin, May 1993.

[Bur92] T.Burd and R.W.Brodersen. Energy Efficient Micorprocessor Design. Kluwer Academic Publishers, January 1992 (1st Edition).

[Cad06] Cadence, Inc. Cadence SoC Encounter User Guide, 2006.

[Cal05] B.Calhoun and A.Chandrakasan. Ultra-dynamic voltage scaling using sub-threshold operation and local voltage dithering in 90nm cmos. *Proc. of ISSCC*, February 2005.

[Car96] M.Carlisle. Olden: Parallelizing programs with dynamic data structures on distributed-memory machines. PhD Thesis, Princeton University Department of Computer, 1996.

[Cas05] F.Castro, D.Chaver, L.Pinuel, M.Prieto, F.Tirado, and M.Huang. Load-store queue management: an energy-efficient design based on a state-filtering mechanism. pages 617–624, Oct 2005.

[Cat98a] F.Catthoor, D.Verkest, and E.Brockmeyer. Proposal for unified system design meta flow in task-level and instruction-level design technology research for multi-media applications. *Proc. of 11th Intnl. Symposium on System Synthesis (ISSS)*, pages 89–95, 1998.

[Cat98b] F.Catthoor, S.Wuytack, E.De Greef, F.Balasa, L.Nachtergaele, and A.Vandecappelle. Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design. Kluwer Acad Publ. Boston, 1998.

[Cat99] F.Catthoor. Energy-delay efficient data storage and transfer architectures and methodologies: Current solutions and remaining problems. *Journal of VLSI Signal Processing*, 21:219–231, 1999.

[Cat02] F.Catthoor, K.Danckaert, C.Kulkarni, E.Brockmeyer, P.G.Kjeldsberg, T.Van Achteren, T.Omnes, "Data access and storage management for embedded programmable processors", ISBN 0-7923-7689-7, Kluwer Acad. Publ., Boston, 2002.

[Cat09] F.Catthoor, "Back- and foreground memory organisation for energy-efficient data parallel access", IMEC Internal report, Feb. 2009.

[Cha82] G.Chaitin. Register allocation and spilling via graph coloring. *Proc. of Compiler Construction*, 1982.

[Cha00] N.Chang, K.Kim, and H.Lee. Cycle-accurate energy consumption measurement and analysis: case study of arm7tdmi. *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, pages 185–190, 2000.

[Che04] S.Cherry. Edholm's law of bandwidth. *IEEE Trans. Spectrum*, Vol.41, pages 58–60, July 2004.

[Che05] A.Cheng and G.Tyson. An energy efficient instruction set synthesis framework for low power embedded system designs. *IEEE Trans. Comput.*, 54(6):698–713, 2005.

[Che06] G.Chen, O.Ozturk, M.Kandemir, and M.Karakoy. Dynamic scratch-pad memory management for irregular array access patterns. *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 931–936, 2006.

[Chi00] D.Chiou, P.Jain, L.Rudolph, and S.Devadas. Application-specific memory management for embedded systems using software-controlled caches. *DAC '00: Proceedings of the 37th conference on Design automation*, pages 416–419, New York, NY, USA, 2000. ACM Press.

[Cma99] R.Cmar, L.Rijnders, P.Schaumont, S.Vernalde, and I.Bolsens. A methodology and design environment for DSP ASIC fixed point refinement. *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, pages 271–276, 1999.

[Coh05] A.Cohen, M.Sigler, S.Girbal, O.Temam, D.Parello, and N.Vasilache. Facilitating the search for compositions of program transformations. *Proc. of ICS*, pages 151–160, 2005.

[Col03] O.Colavin and D.Rizzo. A scalable wide-issue clustered VLIW with a reconfigurable interconnect. *CASES '03: Proceedings of the 2003 international conference on Compilers, architectures and synthesis for embedded systems*, pages 148–158. ACM Press, 2003.

[Con96a] T.Conte, S.Banerjia, S.Larin, and K.Menezes. Instruction fetch mechanisms for vliw architectures with compressed encodings. *Proc. of 29th Intnl. Symposium on Microarchitecture (MICRO)*, December 1996.

[Coo01] K.Cooper, L.Simpson, and C.Vick. Operator strength reduction. *ACM Trans. Program. Lang. Syst.*, 23(5):603–625, 2001.

[Cor98] T.Cormen, C.E. Leicerson, and R.Rivest. Introduction to Algorithms. Prentice Hall, 1998.

[Corp98] H.Corporaal. Microprocessor Architectures : From VLIW to TTA. John Wiley & Sons, 1998.

[Cos07] S.Cosemans, W.Dehaene, and F.Catthoor. A low power embedded sram for wireless applications. *IEEE J. of Solid-state Circ.*, Vol.SC-42 ,pages 1607–1617, July 2007.

[Cos08] S.Cosemans, W.Dehaene, and F.Catthoor. A 3.6pj/access 480mhz, 128kbit on-chip sram with 850mhz boost mode in 90nm cmos with tunable sense amplifiers to cope with variability. *Proc. 34th Europ. Solid-State Circuits Conf.*, ESSCIRC, Edinburgh, UK, pp.278–281, Sep. 2008.

[Cot02] S.Cotterell and F.Vahid. Synthesis of customized loop caches for core-based embedded systems. *Proc. of Intnl. Conf. on Computer Aided Design (ICCAD)*, November 2002.

[CoW08a] CoWare Inc., www.coware.com/products/processordesigner.php. CoWare Processor Designer, 2008.

[CoW08b] CoWare Inc., http://www.coware.com/products/virtualplatform.php. CoWare Virtual Platform, 2008.

[Dal01] W.Dally and B.Towles. Route packets, not wires: Interconnect woes through communication-based design. *Proc. of the 38th Design Automation Conf.*, 2001.

[Dal04] W.Dally, U.Kapasi, B.Khailany, J.Ahn, , and A.Das. Stream processors: Programmability with efficiency. *ACM Queue*, 2(1), March 2004.

[Dal05] W.Dally. Low power architectures. *IEEE Intnl. Solid State Circuits Conf., Panel Talk on "When Processors Hit the Power Wall"*, February 2005.

[Das05] M.Dasygenis, E.Brockmeyer, B.Durinck, F.Catthoor, D.Soudris, and A.Thanailakis. A memory hierarchical layer assigning and prefetching technique to overcome the memory performance/energy bottleneck. *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 946–947, Washington, DC, USA, 2005. IEEE Computer Society.

[Das06] A.Das, W.Dally, and P.Mattson. Compiling for stream processing. *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 33–42, New York, NY, USA, 2006. ACM.

[Deb00] S.Debray, W.Evans, R.Muth, and B.De Sutter. Compiler techniques for code compaction. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 22(2):378–415, March 2000.

[DeM05] H.DeMan. Ambient intelligence: Giga-scale dreams and nano-scale realities. *Proc. of ISSCC, Keynote Speech*, February 2005.

[DeM06] G.De Micheli and L.Benini. Networks on Chips: Technology and Tools (Systems on Silicon). Morgan Kaufmann, 2006.

[Der06] J.Derby, R.Montoye, and J.Moreira. Victoria - vmx indirect compute technology oriented towareds in-line acceleration. *Proc. of CF*, pages 303–311, May 2006.

[Dha79] D.Dhamdhere. On algorithms for operator strength reduction. *Communications of ACM* 22(5), 311–312, May 1979.

[Dom05] J.Domelevo. Working on the design of a customizable ultra-low power processor: A few experiments. Master's thesis, ENS Cachan Bretage and IMEC, Sep 2005.

[Domi05] A.Dominguez, S.Udayakumaran, and R.Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing*, Cambridge Publishing, 2005.

[DPG05] RWTH Aachen – University of Technology, http://www.eecs.rwth-aachen.de/dpg/info.html. DPG User Manual Version 2.8, October 2005.

[DWT] Discrete biorthogonal CDF 9/7 wavelet forward and inverse transform (lifting implementation). http://www.ebi.ac.uk/~gpau/misc/dwt97.c.

[Ebe96] C.Ebeling, D.Cronquist, and P.Franklin. Rapid - reconfigurable pipelined datapath. *The 6th Intnl. Workshop on Field-Programmable Logic and Applications*, 1996.

[EDA05] EDA Meister, http://www.eda-meister.org. ASIP Meister, 2005.

[Eka04] V.Ekanayake, I.Kelly, and R.Manohar. An ultra low-power processor for sensor networks. *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 27–36, New York, NY, USA, 2004. ACM Press.

[Eva95] P.Evans, R.Franzon. Energy consumption modeling and optimization for SRAM's. *IEEE Journal of Solid-State Circuits*, Vol.30, pages 571–579, May 1995.

[Eve01] Everything2, http://everything2.com/e2node/Arbitrary-Sized Arbitrary-Sized Software SIMD, 2001.

[Eyr99] J.Eyre and J.Bier. Infineon's tricore tackles dsp. Article 13/5, Microprocessor Report, April 1999.

[Far00] P.Faraboschi, G.Brown, J.Fischer, G.Desoli, and F.Homewood. Lx: A technology platform for customizable VLIW embedded processing. *Proc. of 27th Intnl. Symposium on Computer Architecture (ISCA)*, June 2000.

[Far07] Faraday Technology Corporation, http://www.faraday-tech.com. Faraday UMC 90nm RVT Standard Cell Library, 2007.

[Fei03] Y.Fei, S.Ravi, A.Raghunathan, and N.Jha. Energy estimation for extensible processors. *IEEE Design and Test in Europe Conf. (DATE)*, 2003.

[Fer97] M.Fernandes, J.Llosa, and N.Topham. Using queues for register file organization in vliw architectures. Internal Report ECS-CSG-29-29, Depart of Computer Science, University of Edinburgh, February 1997.

[Fis05] J.Fisher, P.Faraboschi, and C.Young. Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools. Morgan Kauffman, 2005.

[Fix02] J.Fix. Cache performance analysis of algorithms. PhD Dissertation, University of Washington, pages 22–36, 2002.

[Fla02] K.Flautner, N.Kim, D.Blaauw, and T.Mudge. Drowsy caches: Simple techniques for reducing leakage power. *IEEE Symposium on Computer Architecture*, 2002.

[Fly99] M.Flynn, P.Hung, and K.Rudd. Deep-submicron microprocessor design issues. *IEEE MICRO*, 19(4), July–August 1999.

[Fol96] P.Foley. The mpact media processor redefines the multimedia pc. *Compcon '96. 'Technologies for the Information Superhighway' Digest of Papers*, pages 311–318, Feb 1996.

[Fra93] M.Franklin. The multiscalar architecture. Phd dessertation, University of Wisconsin Madison, November 1993.

[Fre] Freescale Semiconductor, http://www.freescale.com/files/32bit/doc/\ref_manual/MPC7400UM.pdf?srch=1. Altivec Velocity Engine.

[Gan07] A.Gangawar, M.Balakrishnan, and A.Kumar. Impact of intercluster communication mechanisms on ilp in clustered VLIW architectures. *ACM TODAES*, Vol.12, No.1, pages 1–29, Jan. 2007.

[GCC07] GCC, the GNU Compiler Collection. http://gcc.gnu.org, 2007.

[Gee09] P.Geens, "Active supply control in static random-access memories", Doctoral dissertation, ESAT/EE Dept., K.U.Leuven, Belgium, April 2009.

[Gem02] T.Gemmeke, M.Gansen, and T.G. Noll. Implementation of scalable power and area efficient high-throughput viterbi decoders. *IEEE Journal of Solid-State Circuits*, volume 37(7), July 2002.

[Ghe09] S.Gheorghita, M.Palkovic, J.Hamers, A.Vandecappelle, S.Mamagkakis, T.Basten, L.Eeckhout, H.Corporaal, F.Catthoor, F.Vandeputte, and K.De Bosschere. System-scenario-based design of dynamic embedded systems. *ACM Trans. on Design Automation of Electronic Systems*, 14(1):1–45, 2009.

[Gir06] S.Girbal, N.Vasilache, C.Bastoul, A.Cohen, D.Parello, M.Sigler, and O.Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intnl. Journal of Parallel Programming*, pages 261–317, October 2006.

[Glo04] J.Glossner, K.Chirca, M.Schulte, H.Wang, N.Nasimzada, D.Har, S.Wang, A.Hoane, G.Nacer, M.Moudgill1, S. Vassiliadis, "Sandblaster Low Power DSP", *Proc. IEEE Custom Integrated Circuits Conf. (CICC)*, Orlando FL, pp.575–581, Sep. 2004.

[Gom04] J.Gómez, P.Marchal, S.Verdoorlaege, L.Piñuel, and F.Catthoor. Optimizing the memory bandwidth with loop morphing. *Proc. of ASAP wsh*, pages 213–223, 2004.

[Gon02] R.Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, volume 20(2), 2002.

[Goo95] G.Goossens, D.Lanneer, M.Pauwels, F.Depuydt, K.Schoofs, A.Kifli, M.Cornero, P.Petroni, F.Catthoor, H.De Man, "Integration of medium-throughput signal processing algorithms on flexible instruction-set architectures", *J. of VLSI signal processing*, special issue on "Design environments for DSP" (eds.I.Verbauwhede, J.Rabaey), No.9, Kluwer, Boston, pp.49–65, Jan. 1995.

[Gor02a] A.Gordon-Ross, S.Cotterell, and F.Vahid. Exploiting fixed programs in embedded systems: A loop cache example. *Proc. of IEEE Computer Architecture Letters*, Jan 2002.

[Gor02b] A.Gordon-Ross and F.Vahid. Dynamic loop caching meets preloaded loop caching – a hybrid approach. *Proc. of Intnl. Conf. on Computer Design (ICCD)*, September 2002.

[Guo08] J.Guo. Analysis and Optimization of intra-tile Communication Network. PhD thesis, ESAT/EE Dept., K.U.Leuven, August 2008.

[Hal02] A.Halambi, A.Shrivastava, P.Biswas, N.Dutt, and A.Nicolau. An efficient compiler technique for code size reduction using reduced bit-width isas. *Proc. of Design Automation Conf. (DAC)*, March 2002.

[Har91] R.Hartley. Optimization of canonic signed digit multipliers for filter design. *Proc. of ISCA '91*, pages 1992–1995, June 1991.

[Has96] R.Hashemian. A new method for conversion of a 2's complement to canonic signed digit number system and its representation. *Signals, Systems and Computers*, pages 904–907, 1996.

[Hea91] S.Healey. An algorithm for improving optimal placement for river-routing. *EURO-DAC '91: Proceedings of the conference on European design automation*, pages 232–236, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[Hel02] D.Helms, E.Schmidt, A.Schulz, A.Stammermann, and W.Nebel. An improved power macro-model for arithmetic datapath components. *Proceedings of PATMOS*, pages 359–372, 2002.

[Hen96] J.Hennessy and D.Patterson. Computer Architecture: A Quantitative Approach. Morgan Kauffman, 1996 (Second Edition).

[Het02] S.Hettiaratchi, P.Cheung, and T.Clarke. Performance-area trade-off of address generators for address decoder-decoupled memory. *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 902, Washington, DC, USA, 2002. IEEE Computer Society.

[Hew00] R.Hewlitt and E.Swartzlander Jr. Canonical signed ditgit representation for fir digital filters. *Proc. IEEE Workshop on Signal Processing Systems*, pages 416–426, 2000.

[Hey09] K.Heyrman. Control of Sectioned On-Chip Communication. Doctoral dissertation, CS Dept., U.Gent, Belgium, June 2009.

[Hsi97] M.Hsiao and J.Patel. Effects of delay models on peak power estimation of vlsi sequential circuits. *Proc. Intnl. Conf. on Computer Aided Design*, pages 45–51, 1997.

[Hos04] M.Hosemann, C.Gordon, P.Robelly, H.Seidel, T.Dräge, T.Richter, M.Bronzel, G.Fettweis, "Implementing A receiver for terrestrial DVB in software on an Application specific DSP", *Proc. IEEE Wsh. on Signal Processing Systems (SIPS)*, Austin TX, IEEE Press, pp.53–58, Oct. 2004.

[Hu07] Q.Hu. Hierarchical memory size estimation for loop transformation and data memory platform exploration. PhD thesis, N.Technical Univ. Norway, Trondheim, April 2007.

[Hul08] J.Hulzink, "Wireless and biomedical ASIP experiments", IMEC-WATS, Eindhoven, autumn 2008.

[IBM05] IBM, http://www.research.ibm.com/cell/. The Cell Microprocessor, 2005.

[IMEC] Inter-university micro electronics center. http://www.imec.be.

[Imp99] Improv Systems, Inc, http://www.improvsys.com. Jazz DSP processor: Product Brief, 1999.

[Int] Intel, http://www.intel.com/support/processors/sb/cs-001650.htm. Streaming SIMD Extension 2 (SSE2).

[Isc03] C.Isci and M.Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. *MICRO 36: Proceedings of the 36th annual IEEE/ACM Intnl. Symposium on Microarchitecture*, pp. 93, 2003.

[Iss04] I.Issenin, E.Brockmeyer, M.Miranda, and N.Dutt. Data reuse analysis technique for software-controlled memory hierarchies. *Design Automation and Test in Europe (DATE)*, pages 202–207, March 2004.

[ITR07] ITRS. Intnl. techology roadmap for semiconductors 2007 edition: Interconnect. Technical report, ITRS, http://www.itrs.net/Links/2007ITRS/2007_Chapters/2007_Interconnect.pdf, 2007.

[Jac00] M.Jacome, G.de Veciana, and V.Lapinskii. Exploring performance tradeoffs for clustered VLIW ASIPs. *Proc. of ICCAD*, Nov 2000.

[Jan95] J.Janssen and H.Corporaal. Partitioned register file for TTAs. *Proc. of Micro*, pages 303–312, 1995.

[Jay02a] M.Jayapala, F.Barat, T.Vander Aa, F.Catthoor, G.Deconinck, and H.Corporaal. Clustered l0 buffer organization for low energy embedded processors. *Proc. of 1st Workshop on Application Specific Processors (WASP), held in conjunction with MICRO-35*, November 2002.

[Jay02b] M.Jayapala, F.Barat, P.Op de Beeck, F.Catthoor, G.Deconinck, and H.Corporaal. A low energy clustered instruction memory hierarchy for long instruction word processors. *Proc. of 12th Intnl. Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS)*, September 2002.

[Jay04] N.Jayasena, M.Erez, J.Anh, and W.Dally. Stream register files with indexed access. *Proc. of HPCA*, pages 60–72, February 2004.

[Jay05a] M.Jayapala. Low Energy Instruction Memory Organization. Doctoral dissertation, ESAT/EE Dept., K.U.Leuven, Belgium, Sep. 2005.

[Jay05b] M.Jayapala, F.Barat, T.Vander Aa, F.Catthoor, H.Corporaal, and G.Deconinck. Clustered loop buffer organization for low energy VLIW embedded processors. *IEEE Trans. on Computers*, 54(6):672–683, June 2005.

[Jay08] Murali Jayapala, Praveen Raghavan, Francky Catthoor, Distributed loop controller architecture for multi-threading in uni-threaded processors. Granted Patent US 2008/0294882 A1, Nov. 2008.

[Jel04] P.Jelenkovic and A.Radovanovic. Least-recently-used caching with dependent requests. *Theoretical Computer Science*, 326(1–3):293–327, 2004.

[Jos06] M.Joshi, NS. Nagaraj, and A.Hill. Impact of interconnect scaling and process variations on performance. *Proc. of CMOS Emerging Technologies*, 2006.

[Jou90] N.Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *Proc. of Intnl. Symposium on Computer Architecture (ISCA)*, May 1990.

[Jou94] N.Jouppi and S.Wilton. Trade-offs in two level on-chip caching. *Proc. of Intnl. Symposium on Computer Architecture (ISCA)*, May 1994.

[JPEG] JPEG 2000 Image Coding Standard. www.jpeg.org/jpeg2000/.

[Jul03] N.Julien, J.Laurent, E.Senn, and E.Martin. Power consumption modeling and characterization of the ti c6201. *IEEE Micro*, Sep–Oct 2003.

[Kad02] I.Kadayif and M.Kandemir. Instruction compression and encoding for low-power systems. *IEEE Conf on ASIC/SOC*, pages 301–305, 2002.

[Kan99] M.Kandemir, J.Ramanujan, and A.Chowdhury. Improving cache locality by a combination of loop and data transformation. *IEEE Trans. on Computers*, 48(2), 1999.

[Kan01a] M.Kandemir, I.Kadayif, and U.Sezer. Exploiting scratch-pad memory using preseburger formulas. *Intnl. Symposium on System Synthesis (ISSS)*, 7(12), 2001.

[Kan01b] M.Kandemir, J.Ramanujam, J.Irwin, N.Vijaykrishnan, I.Kadayif, and A.Parikh. Dynamic management of scratch-pad memory space. *DAC '01: Proceedings of the 38th conference on Design automation*, pages 690–695, New York, NY, USA, 2001. ACM Press.

[Kan04a] M.Kandemir, I.Kadayif, A.Choudhary, J.Ramanujam, and I.Kolcu. Compiler-directed scratch pad memory optimization for embedded multiprocessors. *IEEE Trans on VLSI*, pages 281–287, March 2004.

[Kan04b] M.Kandemir and J.Ramanujan. A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, 23(2):243–259, March 2004.

[Kap03] U.Kapasi, S.Rixner, W.Dally, B.Khailany, J.Ahn, P.Mattson, and J.Owens. Programmable stream processors. *IEEE Computer*, pages 54–62, Aug. 2003.

[Kar04] T.Karalar, S.Yamashita, M.Sheets, and J.Rabaey. An integrated, low power localization system for sensor networks. *MobiQuitous*, pages 24–30, 2004.

[Kar07] G.Karakonstantis and K.Roy. An optimal algorithm for low power multiplierless fir filter design using chebychev criterion. *IEEE Intnl. Conf. on Acoustics, Speech and Signal Processing, ICASSP 2007*, 2:II–49–II–52, April 2007.

[Kat00] V.Kathail, M.Schlansker, and B.Rau. Hpl-pd architecture specification: Version 1.1. Technical Report HPL-93-80 (R.1), HP Research Labs, USA, 2000.

[Kav05] N.Kavvadias and S.Nikolaidis. Zero-overhead loop controller that implements multimedia algorithms, July 2005.

[Kax01] S.Kaxiras, G.Narlikar, A.Berenbaum, and Z.Hu. Comparing power consumption of an SMT and a CMP DSP for mobile phone workloads. *Proc. of Intnl. Conf. on Compilers, Architecture, and Synthesis for Smbedded Systems (CASES)*, pages 211–220, November 2001.

[Keu00] K.Keutzer, S.Malik, A.Newton, J.Rabaey, and A.Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 19(12):1523–1543, December 2000.

[Kim01] S.Kim, N.Vijaykrishnan, M.Kandemir, A.Sivasubramaniam, M.Irwin, and E.Geethanjali. Power-aware partitioned cache architectures. *Proc. of ACM/IEEE Intnl. Symposium on Low Power Electronics (ISLPED)*, August 2001.

[Kim04] N.Kim, K.Flautner, D.Blaauw, and T.Mudge. Circuit and microarchitectural techniques for reducing cache leakage power. *IEEE Trans on VLSI*, pages 167–184, 2004.

[Kim05] H.Kim and H.Oh. A low-power dsp-enhanced 32-bit eisc processor. *Proc. of HiPEAC*, pages 302–316, 2005.

[Kin71] W.King. Analysis of paging algorithm. *Proc. of IFIP Congress*, pages 485–490, August 1971.

[Kin00] J.Kin, M.Gupta, and W.Mangione-Smith. Filtering memory references to increase energy efficiency. *IEEE Trans. on Computers*, 49(1):1–15, January 2000.

[Kje01] P.Kjeldsberg. Storage requirement estimation and optimisation for data-intensive applications. PhD thesis, Norwegian Univ. of Science and Technology, Trondheim, Norway, March 2001.

[Kob07] Y.Kobayashi, M.Jayapala, PS.Raghavan, F.Catthoor, M.Imai. "Methodology for operation scheduling and L0 cluster generation for low energy heterogeneous VLIW processors". *ACM Trans. on Design Automation for Embedded Systems (TODAES)*, Vol.12, No.4, Article 41, pp.1–28, Sep. 2007.

[Kob07b] Y.Kobayashi. Low Power Design Method for Embedded Systems Using VLIW Processor. PhD thesis, Graduate School of Inforamation Science and Technology at Osaka University, July 2007.

[Kog06] T.Kogel, R.Leupers, and H.Meyr. Integrated System-Level Modeling of Network-on-Chip enabled Multi-Processor Platforms. Springer, 2006.

[Koz03] C.Kozyrakis and D.Patterson. Scalable vector processors for embedded systems. *IEEE Micro*, 23(6):36–45, 2003.

[Kra07] S.Kraemer, R.Leupers, G.Ascheid, and H.Meyr. Interactive presentation: Softsimd - exploiting subword parallelism using source code transformations. *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1349–1354, San Jose, CA, USA, 2007. EDA Consortium.

[Kri09] A.Kritikakou. Low cost low energy embedded processor for online biotechnology monitoring applications. Master's thesis, Dept. of Computer Eng. and Informatics, Univ. of Patras and IMEC, Feb. 2009.

[Kuk92] K.Kukich. Technique for automatically correcting words in text. *ACM Comput. Surv.*, 24(4):377–439, 1992.

[Kum08] A.Kumar, K.van Berkel, "Vectorization of Reed solomon decoding and mapping on the EVP", *Proc. 9th ACM/IEEE Design and Test in Europe Conf.(DATE)*, Munich, Germany, pp.450–455, March 2008.

[Lad99] R.Ladner, J.Fix, and A.La Marca. Cache performance analysis of traversals and random accesses. pages 613–622, 1999.

[Laf99] G.Lafruit, F.Catthoor, J.Cornelis, H.De Man, "An efficient VLSI architecture for the 2-D wavelet transform with novel image scan", *IEEE Trans. on VLSI Systems*, Vol.7, No.1, pp.56–68, March 1999.

[Lam04] A.Lambrechts, T.Van der Aa, M.Jayapala, A.Leroy, G.Talavera, A.Shickova, F.Barat, F.Catthoor, D.Verkest, G.Deconinck, H.Coporaal, F.Robert, and J.Bordoll. Design style case study for compute nodes of a heterogenenous noc platform. *25th IEEE Real-Time Systems Symposium (RTSS)*, December 2004.

[Lam05] A.Lambrechts, P.Raghavan, A.Leroy, G.Talavera, T.Van der Aa, M.Jayapala, F.Catthoor, D.Verkest, G.Deconinck, H.Corporaal, F.Robert, J.Carrabina, "Power breakdown analysis for a heterogeneous NoC platform running a video application", *Proc. Intnl. Conf. on Applic.-Spec. Array Processors (ASAP)*, Samos, Greece, pp.179–184, July 2005.

[Lam07] A.Lambrechts, P.Raghavan, D.Novo Bruno, E.Rey Ramos, M.Jayapala, J.Absar, F.Catthoor, D.Verkest, "Enabling word-width aware energy optimizations for embedded processors", *Proc. Intnl. Wsh. on Optimisations for DSP and Embedded Systems (ODES)*, San Jose CA, pp.66–75, March 2007.

[Lam08a] A.Lambrechts, P.Raghavan, M.Jayapala, and F.Catthoor. Method for exploiting heterogeneous word-width information for simd execution. EU Patent Filed, 2008.

[Lam08b] A.Lambrechts, P.Raghavan, M.Jayapala, F.Catthoor, D.Verkest, "Cost-aware strength reduction for constant multiplication in VLIW processors", *Proc. 6th Wsh. on Optim. for DSP and Embedded Systems (ODES)*, Boston MA, pp. 1–8, April 2008.

[Lam09] A.Lambrechts, "Energy-aware datapath optimizations at the architecture-compiler interface", Doctoral dissertation, ESAT/EE Dept., K.U.Leuven, Belgium, June 2009.

[Lap02] V.Lapinskii, M.Jacome, and G.de Veciana. Application-specific clustered vliw datapaths: Early exploration on a parameterized design space. *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, 21(8):889–903, August 2002.

[Lar05] S.Larsen, R.Rabbah, and S.Amarasinghe. Exploiting vector parallelism in software pipelined loops. *MICRO '05: Proceedings of the 38th Annual IEEE/ACM Intnl. Symposium on Microarchitecture (MICRO'05)*, pages 119–129, Washington, DC, USA, 2005. IEEE Computer Society.

[Lee99] L.Lee, W.Moyer, and J.Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. *Proc. of Intnl. Symposium on Low Power Electronic Design (ISLPED)*, August 1999.

[Lee03] J.Lee, K.Choi, and N.Dutt. Energy-efficient instruction set synthesis for application-specific processors. *ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design*, pages 330–333, New York, NY, USA, 2003. ACM Press.

[Lef01] V.Lefévre. Multiplication by an integer constant. Technical report, INRIA, France, May 2001.

[Lei83] C.Leiserson and R.Pinter. Optimal placement for river routing. *SIAM J. Comput.*, 12(3):447–462, 1983.

[Ler06] T.Leroy, E.Vranken, A.Van Brecht, E.Streulens, B.Sonck, D.Berckmans, "A computer vision method for on-line behavioral quantification of individually caged poultry", *Trans. of the ASAE*, Vol.49, No.3, pp.795–802, 2006.

[Ler06b] A.Leroy, "Optimizing the on-chip communication architecture of low power systems-on-chip in deep submicron technology", Doctoral dissertation, U.L.Bruxelles, Belgium, Dec. 2006.

[Ler08] A.Leroy, D.Milojevic, D.Verkest, F.Robert, and F.Catthoor. Concepts and implementation of spatial division multiplexing forguaranteed throughput in networks-on-chip. *IEEE Trans. on Computers*, 57(9):1182–1195, September 2008.

[Li09] M.Li, D.Novo, B.Bougard, T.Carlson, L.Van der Perre, and F.Catthoor. Generic multi-phase software pipelined partial fft on instruction level parallel architectures. *IEEE Trans. on Signal Processing*, Vol.SP-57, No.4, pp.1604–1615, April 2009.

[Lib02] LSF: Liberty Simulation Framework 1.0. http://liberty.princeton.edu/Software/LSE, 2002.

[Liv02] N.Liveris, N.Zervas, D.Soudris, and C.Goutis. A code transformation-based methodology for improving i-cache performance of dsp applications. *Proc. of Design Automation and Test in Europe (DATE)*, March 2002.

[Log04] M.Loghi, F.Angiolini, D.Bertozzi, L.Benini, and R.Zafalon. Analyzing on-chip communication in a mpsoc environment. *IEEE Design and Test in Europe Conf. (DATE)*, 2004.

[Lu99] G.Lu, H.Singh, M.Lee, N.Bagherzadeh, F.Kurdahi, and E.Filho. The MorphoSys parallel reconfigurable system. *Proc. of Euro-Par*, 1999.

[Mah92] S.Mahlke, D.Lin, W.Chen, R.Hank, and R.Bringmann. Effective compiler support for predicated execution using the hyperblock. *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 45–54. IEEE Computer Society Press, 1992.

[Mam04] M.Mamidipaka, K.Khouri, N.Dutt, and M.Abadir. Analytical models for leakage power estimation of memory array structures. *IEEE CODES+ISSS*, pages 146–151, 2004.

[Man10] S.Mamagkakis, D.Atienza, F.Catthoor, D.Soudris, C.Poucet, A.Bartzas, C.Baloukas, M.Peon, and J.Mendias Cuadros. Dynamic memory management for embedded programmable platforms: systematic methodology and practice. Springer, Heidelberg, Germany, Spring 2010.

[Mar03] P.Marwedel. Embedded System Design. Kluwer Academic Publishers (Springer), Norwell, MA, USA, 2003.

[Mat03] S.Mathew, M.Anders, R.Krishnamurthy, and S.Borkar. A 4-ghz 130-nm address generation unit with 32-bit sparse-tree adder core. *IEEE Journal of Solid-State Circuits*, 38(5), may 2003.

[Maz93] C.Mazenc, X.Merrheim, J.Muller, "Computing Functions cos/sup -1/ and sin/sup -1/ Using CORDIC", *IEEE Trans. on Computers*, Vol.C-42, No.1, pp.118–122, Jan. 1993.

[McK08] S.McKeown, R.Woods, and J.McAllister. Power efficient dynamic-range utilisation for dsp on fpga. *SiPS 2008: IEEE Workshop on Signal Processing Systems*, pages 233–238, 2008.

[MedB] Mediabench multi-media application benchmark suite. http://www.cs.ucla.edu/~leec/mediabench.

[Med97] C.Lee, M.Potkonjak and M.Smith, "MediaBench: a Tool for Evaluating and Synthesizing Multimedia and Communication Systems", *IEEE Intnl. Symp. on Microarchitecture*, Research Triangle Park NC, pp.330–335, Dec. 1997.

[Mei02] B.Mei, S.Vernalde, D.Verkest, H.De Man, and R.Lauwereins. DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. *Proc. of Intnl. Conf. on Field Programmable Technology*, pages 166–173, 2002.

[Mei03a] B.Mei, S.Vernalde, D.Verkest, H.De Man, and R.Lauwereins. ADRES: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. *Proc. IEEE Conf. on Field-Programmable Logic and its Applications (FPL)*, pages 61–70, Lisbon, Portugal, September 2003.

[Mei03b] B.Mei, S.Vernalde, D.Verkest, H.De Man, and R.Lauwereins. Exploiting loop-level parallelism for coarse-grained reconfigurable architecture using modulo scheduling. *Proc. of Design, Automation and Test in Europe (DATE)*, 2003.

[Men07] Mentor Graphics, http://www.mentor.com/products/fv/emulation/vstation_pro/. VStationPRO, 2007.

[Mir96] M.Miranda, F.Catthoor, M.Janssen, and H.De Man. Adopt: Efficient hardware address generation in distributed memory architectures. *Proc. 9th ACM/IEEE Intnl. Symp. on System-Level Synthesis (ISSS)*, La Jolla CA, pp.20–25, Nov. 1996.

[Mir97] M.Miranda, M.Kaspar, F.Catthoor, and H.de Man. Architectural exploration and optimization for counter based hardware address generation. *EDTC '97: Proceedings of the 1997 European conference on Design and Test*, page 293, Washington, DC, USA, 1997. IEEE Computer Society.

[Mir98] M.Miranda, F.Catthoor, M.Janssen, H.De Man, "High-level Address Optimisation and Synthesis Techniques for Data-Transfer Intensive Applications", *IEEE Trans. on VLSI Systems*, Vol.6, No.4, pp.677–686, Dec. 1998.

[Mon05] Montium TP Processor, http://www.recoresystems.com. Montium Tile Processor Reference Manual, 2005.

[Moo97] D.Moolenaar, L.Nachtergaele, F.Catthoor, and H.De Man. System-level power exploration for mpeg-2 decoder on embedded cores: a systematic approach. *IEEE Wsh. on Signal Processing Systems*, 1997.

[Muc97] S.Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997.

[Mur09] S.Murali. Designing Reliable and Efficient Networks on Chips. Lecture Notes in Electrical Engineering, Issue No. 34, Springer, 2009.

[Naz05] L.Nazhandali, B.Zhai, J.Olson, A.Reeves, M.Minuth, R.Helfand, S.Pant, T.Austin, and D.Blaauw. Energy optimization of subthreshold-voltage sensor network processors. *Proc. of ISCA*, 2005.

[Nov08] D.Novo, B.Bougard, A.Lambrechts, L.Van der Perre, and F.Catthoor. Scenario-based fixed-point data format refinement to enable energy-scalable software defined radios. *Proc. of Design and Test in Europe Conf.(DATE)*, pages 722–727, March 2008.

[Nov09] D.Novo, B.Bougard, L.Van der Perre, F.Catthoor, "Finite-precision processing in wireless applications", *Proc. 12th ACM/IEEE Design and Test in Europe Conf.(DATE)*, Nice, France, pp.1230–1233, April 2009.

[Nvi09] Nvidia, http://www.nvidia.com/object/geforce_8600M.html. NVidia GEForce 8600 Series Processor, January 2009.

[OdB01] P.Op de Beeck, F.Barat, M.Jayapala, and R.Lauwereins. CRISP: A template for reconfigurable instruction set processors. *Proc. of Intnl. conference on Field Programmable Logic (FPL)*, August 2001.

[OdB03] P.Op de Beeck, C.Ghez, E.Brockmeyer, M.Miranda, F.Catthoor, and G.Deconinck. Background data organisation for the low-power implementation in real-time of a digital audio broadcast receiver on a simd

processor. *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 11144, Washington, DC, USA, 2003. IEEE Computer Society.

[Oze05] E.Ozer and T.M.Conte. High-performance and low-cost dual thread VLIW processor using weld architecture paradigm. *IEEE Trans. on Parallel and Distributed Systems*, volume 16(12), December 2005.

[Oze08] E.Özer, A.Nisbet, and D.Gregg. A stochastic bitwidth estimation technique for compact and low-power custom processors. *Trans. on Embedded Computing Sys.*, 7(3):1–30, 2008.

[PAC03] PACT XPP Technologies, 2003. http://www.pactcorp.com.

[Pal07] M.Palkovic. Enhanced applicability of loop transformations. PhD thesis, T.U. Eindhoven, September 2007.

[Pal05] M.Palkovic, E.Brockmeyer, P.Vanbroekhoven, H.Corporaal, F.Catthoor, "Systematic Preprocessing of Data Dependent Constructs for Embedded Systems", *Proc. IEEE Wsh. on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Leuven, Belgium, Lecture Notes Comp. Sc., Springer-Verlag, Vol.3728, pp.89–90, Sep. 2005.

[Pal97] S.Palacharla, N.Jouppi, and J.Smith. Complexity-effective superscalar processor. *Proc. of Intnl. Symposium on Computer Architecture (ISCA)*, June 1997.

[Pal08] M.Palkovic and A.Folens. Mapping of the 40mhz WLAN SDM receiver on the FLAI ADRES baseband engine. Apollo Deliverable 200803_DE_SDR_BB_D4i, IMEC vzw, April 2008.

[Pan97] P.Panda, N.Dutt, and A.Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. *EDTC '97: Proceedings of the 1997 European conference on Design and Test*, pp.7, Washington, DC, USA, 1997. IEEE Computer Society.

[Pan98] P.Panda, A.Nicolau, and N.Dutt. Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[Pan01] P.Panda, F.Catthoor, N.Dutt, K.Danckaert, E.Brockmeyer, C.Kulkarni, A.Vandercappelle, and P.Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 6(2):149–206, April 2001.

[Pap06] A.Papanikolaou. Application-driven software configuration of communication networks and memory organizations. PhD thesis, CS Dept., U.Gent, Belgium, December 2006.

[Par99] K.Parhi and T.Nishitani. Digital Signal Processing for Multimedia Systems. CRC Publications, 1999.

[Par01] I.Park and H.Kang. Digital filter synthesis based on minimal signed digit representation. *Proc. of DAC 2001*, pages 486–473, 2001.

[Par03] I.Park, C.Ooi, and T.Vijaykumar. Reducing design complexity of the load/store queue. 2003.

[Pat98] D.Patterson and J.Hennessy. Computer Organization and Design: Hardware and Software Interface. Morgan Kauffman, 1998 (Second Edition).

[Pau92] M.Pauwels, Filip Thoen, Eric Beeckmans, Gert Goossens, Francky Catthoor, Hugo De Man, "An Application Specific Multi-precision Multiplier Architecture for a Dynamic Range Compressor for Audio", *Proc. 6th Eur. Signal Processing Conf.*, EUSIPCO-92, Brussels, Belgium, Elsevier Publ., Amsterdam, pp.1569–1572, Aug. 1992.

[Pau95] M.Pauwels, G.Goossens, F.Catthoor, H.De Man, "Formalisation of multi-precision arithmetic for high-level synthesis of DSP architectures", special issue on "VLSI Design Methodologies for DSP Systems", M.Bayoumi (Ed.), *J. of VLSI Signal Processing*, Kluwer, Boston, No.11, Oct. 1995.

[Pau00] P.G.Paulin, "Towards Application-Specific Architecture Platforms: Embedded Systems Design Automation Technologies", *26th Euromicro Conf.*, Maastricht, The Netherlands, pp.1028–, Sep. 2000.

[Pea01] M.Powell and et al. Reducing set-associative cache energy via way-prediction and selective direct-mapping. *Proc. of 34th Intnl. Symposium on Microarchitecture (MICRO)*, November 2001.

[Pet03] P.Petrov and A.Orailoglu. Application-specific instruction memory customizations for power-efficient embedded processors. *IEEE Design and Test*, pages 18–25, 2003.

[Phi] Philips Research, http://www.siliconhive.com. Philips SiliconHive Avispa Accelerator.

[Phi04] S.Phillips, A.Sharma, and S.Hauck. Automating the layout of reconfigurable subsystems via template reduction. *FCCM*, pages 340–341, 2004.

[Pol04] F.Poletti, P.Marchal, D.Atienza, L.Benini, F.Catthoor, and J.M.Mendias. An integrated hardware/software approach for run-time scratchpad management. *DAC*, pages 238–243, 2004.

[Pon02] D.Ponomarev, G.Kucuk, and K.Ghose. Accupower: An accurate power estimation tool for superscalar microprocessors. *Proc. of DATE*, pages 124–130, 2002.

[Por06] T.Portero, G.Talavera, F.Catthoor, J.Carrabina, "A study of a MPEG-4 codec in a Multiprocessor platform", *Proc. Intnl. Symp. on Industrial Elec.(ISTE)*, Montreal, Canada, pp.661–666, July 2006.

[Poz07] L.Pozzi, P.G.Paulin, "A future of customizable processors: are we there yet?", *Proc. 10th ACM/IEEE Design and Test in Europe Conf.(DATE)*, Nice, France, pp.1224–1225, April 2007.

[Psy10] G.Psychou. Optimized SIMD scheduling and architecture implementation for ultra-low energy bioimaging processor. Master's thesis, Dept. of Computer Eng. and Informatics, Univ. of Patras and IMEC, March 2010.

[Qui00] F.Quillere, S.Rajopadhye, and D.Wilde. Generation of efficient nested loops from polyhedra. *Intl. Journal on Parallel Programming*, 2000.

[Rab0-] J.Rabaey, U.C.Berkeley, sensor node publications at website http://www.eecs.berkeley.edu/Pubs/Faculty/rabaey.html.

[Rab04] R.Rabbah, I.Bratt, K.Asanovic, and A.Agarwal. Versatility and versabench: A new metric and a benchmark suite for flexible architectures. June 2004.

[Rag06a] P.Raghavan and F.Catthoor. Ultra low power asip (application-domain specific instruction-set processor) micro-computer. EU Patent Filed EP 1 701 250 A1, September 2006.

[Rag06b] P.Raghavan, A.Lambrechts, M.Jayapala, F.Catthoor, and D.Verkest. "Low power custom crossbars in SIMD shufflers", *Proc. 6th Symp. on Program acceleration by Application-driven and architecture-driven Code Transf. (ACES)*, Edegem, Belgium, Oct. 2006.

[Rag07a] P.Raghavan, A.Lambrechts, M.Jayapala, F.Catthoor, D.Verkest, H.Corporaal, "Very wide register: an asymmetric register file organisation for low power embedded processors", *Proc. 10th ACM/IEEE Design and Test in Europe Conf.(DATE)*, Nice, France, pp.1066–1071, April 2007.

[Rag07b] P.Raghavan, N.Sethubalasubramanian, S.Munaga, E.Rey Ramos, M.Jayapala, D.Weiss, F.Catthoor, D.Verkest, "Semi Custom Design: A Case Study on SIMD Shufflers", *Proc. IEEE Wsh. on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Lecture Notes in Computer Science Vol.4644, Goteborg, Sweden, pp.433–442, Sep. 2007.

[Rag07c] P.Raghavan, S.Munaga, E.Rey Ramos, A.Lambrechts, M.Jayapala, F.Catthoor, D.Verkest, "A Customized Cross-bar for Data-shuffling in Domain-specific SIMD Processors", *Proc. Intnl. Wsh. on Architecture of Computing Systems (ARCS)*, Zurich, Switzerland, pp.57–68, March 2007.

[Rag08a] Praveen Raghavan, Murali Jayapala, Francky Catthoor, Absar Javed, and Andy Lambrechts. Method and system for automated code conversion. Granted Patent US 2008/0263530 A1, Oct. 2008.

[Rag08b] P.Raghavan, A.Lambrechts, J.Absar, , M.Jayapala, and F.Catthoor. COFFEE: COmpiler Framework For Energy-aware Expoloration. *Proc. Intnl. Conf. on High-Perf. Emb. Arch. and Compilers (HIPEAC'08)*, Goteborg, Sweden, pp.193–208, Jan. 2008.

[Rag09a] P.Raghavan, A.Lambrechts, M.Jayapala, F.Catthoor, D.Verkest, "EMPIRE: Empirical Power/Area/Timing Models for Register Files", *Microprocessors and Microsystems J.*, on-line publ. Feb. 2009.

[Rag09b] P.Raghavan, "Low energy VLIW architecture extensions and compiler plug-ins for embedded systems", Doctoral dissertation, ESAT/EE Dept., K.U.Leuven, Belgium, June 2009.

[Rag09c] P.Raghavan, A.Lambrechts, M.Jayapala, F.Catthoor, D.Verkest, "Distributed loop controller for multi-threading in uni-threaded ILP architectures", *IEEE Trans. on Computers*, Vol.58, No.3, pp.311–321, March 2009.

[Ram04] F.Rampogna, P.Pfister, C.Arm, P.Volet, J.Masgonty, C.Piguet, "Macgic, a low-power reconfigurable DSP", book chapter in "Low Power Electronics Design", (eds. C.Piguet e.a.), CRC Press, April 2004.

[Ram05] A.Ramachandran and M.Jacome. Energy-delay efficient data memory subsystems. *IEEE Signal Processing Magazine*, pages 23–37, May 2005.

[Ram07] U.Ramacher, "Software-Defined Radio Prospects for Multistandard Mobile Phones", *IEEE Computer Magazine*, Vol.40, No.10, pp.62–69, Oct. 2007.

[Rao78] G.Rao. Performance analysis of cache memories. *J. ACM*, 25(3):378–395, 1978.

[Rau94] B.Rau. Iterative modulo scheduling. Technical Report HPL-94–115, HP Laboratories, 1994.

[Rix00a] S.Rixner, W.Dally, B.Khialany, P.Mattson, U.Kapasi, and J.Owens. Register organization for media processing. *Proc. of 26th Intnl. Symposium on High-Performance Computer Architecture (HiPC)*, pages 375–386, January 2000.

[Roc02] L.Rocha, L.Velho, P.Carvalho, "Image moments-based structuring and tracking of objects", *Proc. IEEE Braz. Symp. on Computer Graphics and Image Processing*, pp.99–105, Oct. 2002.

[Roc06] R.Rocher, D.Menard, N.Herv, and O.Sentieys. Fixed-point configurable hardware components. EURASIP Journal on Embedded Systems, Article ID 23197, 2006.

[Ron07] H.Rong, Z.Tang, R.Govindarajan, A.Douillet, and G.Gao. Single-dimension software pipelining for multidimensional loops. *ACM Trans. Archit. Code Optim.*, 4(1), 2007.

[Rot96] E.Roternberg, S.Bennett, and J.Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. *Proc. of 29th Intnl. Symposium on Microarchitecture (MICRO)*, December 1996.

[Ryu01] K.Ryu, E.Shin, and V.Mooney. A comparison of five different multiprocessor soc bus architectures. *Proc. of the EUROMICRO Symposium on Digital Systems Design (EUROMICRO'01)*, pages 202–209, 2001.

[Sandbridge] Sandbridge processors, SB35x and Sandblaster, http://www.sandbridgetech.com/technology.php

[San01] J.Sánchez and A.González. Modulo scheduling for a fully-distributed clustered VLIW architectures. *Proc. of 29th Intnl. Symposium on Microarchitecture (MICRO)*, December 2001.

[Sas95] R.Sastry and K.Rajgopal. Ecg compression using wavelet transform. *IEEE Proc. 14th Conf. of the Biomedical Engineering Society of India*, pages 2/52–2/53, Feb 1995.

[Sat91] J.Sato, M.Imai, T.Hakata, A.Alomary, N.Hikichi, "An integrated design environment for application-specific integrated processor", *Proc. IEEE Intnl. Conf. Computer Design*, Rochester NY, pp.106–111, Oct. 1991.

[Sca06] D.Scarpazza, P.Raghavan, D.Novo, F.Catthoor, and D.Verkest. Software simultaneous multi-threading, a technique to exploit task-level parallelism to improve instruction- and data-level parallelism. *Proc. of PATMOS*. Springer Verlag LNCS, Sep. 2006.

[Sch04] M.Schneider, H.Blume, and T.Noll. Power estimation on functional level for programmable processors. *Advances in Radio Science*, 2:215–219, May 2004.

[Sch07] T.Schuster, B.Bougard, P.Raghavan, R.Priewasser, D.Novo, L.Vanderperre, and F.Catthoor. Design of a low power presynchronization asip for multimode sdr terminals. *Proc. of SAMOS*, 2007.

[Schl07] O.Schliebusch, H.Meyr, R.Leupers, "Optimized ASIP Synthesis from Architecture Description Language Models", ISBN 978-1-4020-5685-7, Springer, Heidelberg, Germany, 2007.

[Set07] S.Sethumadhavan, F.Roesner, J.Emer, D.Burger, and S.Keckler. Late-binding: enabling unordered load-store queues. pages 347–357, 2007.

[Shi99] W.Shiue and C.Chakrabarti. Memory exploration for low power embedded systems. *Proc. of Design Automation Conf. (DAC)*, June 1999.

[Shi01] P.Shivakumar and N.Jouppi. CACTI3.0: A integrated cache timing, power, and area model. Technical report, COMPAQ Western Research Laboratory, Aug. 2001.

[Shi03] A.Shickova, T.Marescaux, D.Verkest, F.Catthoor, S.Vernalde, and R.Lauwereins. Architecture exploration of interconnection networks as a communication layer for reconfigurable systems. *ProRISC*, 2003.

[Sia01] J.Sias, H.Hunter, and W.Hwu. Enhancing loop buffering of media and telecommunications applications using low-overhead predication. *Proc. of 34th Annual Intnl. Symposium on Microarchitecture (MICRO)*, December 2001.

[SilH] Silicon Hive, http://www.siliconhive.com. SiliconHive HiveFlex XSP.

[Sin92] J.Singh, H.Stone, and D.Thibaut. A model of workloads and its use in miss-rate prediction for fully associative caches. *IEEE Trans. on Computers*, 41(7):811–825, 1992.

[Sin00] H.Singh, M.Lee, G.Lu, N.Bagherzadeh, F.Kurdahi, and E.Chaves Filho. Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Comput.*, 49(5): 465–481, 2000.

[Sin01] A.Sinha and A.Chandrakasan. Jouletrack - a web based tool for software energy profiling. *Proc. of Design Automation Conf. (DAC)*, June 2001.

[Smi04] M.Smith, N.Ramsey, and G.Holloway. A generalized algorithm for graph-coloring register allocation. *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 277–288, New York, NY, USA, 2004.

[Sta00] Starcore DSP Techology, http://www.starcore-dsp.com. SC140 DSP Core Reference Manual, June 2000.

[Ste02] S.Steinke, L.Wehmeyer, B.Lee, and P.Marwedel. Assigning program and data objects to scratchpad for energy reduction. *Design Automation and Test in Europe (DATE)*, pages 409–414, March 2002.

[STM00] STMicroelectronics, http://www.st.com. ST120 DSP-MCU Programming Manual, December 2000.

[Sto98] P.Stobach. A new technique in scene adpative coding. *European Signal Processing Conf. (EUSIPCO)*, 1998.

[Sub06] S.Subramaniam and G.Loh. Fire-and-forget: Load/store scheduling with no store queue at all. pages 273–284, 2006.

[Sui01] SUIF2 Compiler System. http://suif.stanford.edu, 2001.

[Syl99] D.Sylvester and K.Keutzer. Getting to the bottom of deep submicron ii: a global wiring paradigm. *ISPD '99: Proceedings of the 1999 international symposium on Physical design*, pages 193–200, New York, NY, USA, 1999. ACM.

[Syn06a] Synopsys, Inc. Design Compiler User Guide, 2006.

[Syn06b] Synopsys, Inc. Prime Power User Guide, 2006.

[Syn08] Synfora Inc., http://www.synfora.com. PICO Express, 2008.

[Tal03] S.Tallam and R.Gupta. Bitwidth aware global register allocation. *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 85–96, New York, NY, USA, 2003. ACM.

[Tal08] G.Talavera, M.Jayapala, J.Carrabina, F.Catthoor, "Address Generation Optimization for Embedded High-Performance Processors: a Survey", *J. of Signal Processing Systems*, Springer, May 2008 (on-line); Vol.53, No.3, pp.271–284, Sep. 2008.

[Tan01] W.Tang, R.Gupta, and A.Nicolau. Design of a predictive filter cache for energy savings in high performance processor architectures. *Proc. of Intnl. Conf. on Computer Design (ICCD)*, Sep. 2001.

[Tan02] W.Tang, R.Gupta, and A.Nicolau. Reducing power with an l0 instruction cache using history-based prediction. *Proc. of Intnl. Workshop on Innovative Architecture for Future Generatio High-Performance processors and Systems (IWIA)*, Jan. 2002.

[Tan08] I.Taniguchi. Systematic Architecture Exploration Method for Low Energy and High Performance Reconfigurable Processors. PhD thesis, Osaka University, Osaka, Japan, Dec. 2009.

[Tan09] I.Taniguchi, M.Jayapala, P.Raghavan, F.Catthoor, K.Sakanushi, Y.Takeuchi, M.Imai. "Systematic Architecture Exploration based on Optimistic Cycle Estimation for Low Energy Embedded Processors". *14th Proc. IEEE Asia and South Pacific Design Autom. Conf.(ASPDAC)*, Yokahama, Japan, pp.449–454, Jan. 2009.

[Tar08] Target, http://www.retarget.com. IP Designer, 2008.

[Thu08] M.Thuresson, L.Spracklen, and P.Stenstrom. Memory-link compression schemes: A value locality perspective. *IEEE Trans. on Computers*, 57(7):916–927, July 2008.

[TI99] Texas Instruments Inc., http://www.ti.com. TMS320C6000 Power Consumption Summary, November 1999.

[TI00] Texas Instruments, Inc, http://www.ti.com. TMS320C6000 CPU and Instruction Set Reference Guide, October 2000.

[TI04] Texas Instruments, http://focus.ti.com/docs/prod/folders/print/tms320c6204.html. TI TMS320C6204 DSP processor, March 2004.

[TI06] Texas Instruments, Inc, http://www.ti.com/. TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide, May 2006.

[TI09a] Texas Instruments, Inc, http://www.ti.com/. MSP430 Ultra Low Power Microcontrollers, January 2009.

[TI09b] Texas Instruments, Inc, http://www.ti.com/. TSM320C54x DSP devices, January 2009.

[TI09c] Texas Instruments, Inc, http://focus.ti.com/dsp/docs/dsphome.tsp?sectionId=46 OMAP and Da Vinci DSP devices, 2009.

[TI09d] TI DSP Benchmark Suite. http://focus.ti.com/docs/toolsw/folders/print/sprc092.html, 2009.

[Tiw94] V.Tiwari, S.Malik, and A.Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, 1994.

[Tiw96] V.Tiwari, S.Malik, A.Wolfe, and M.Lee. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, pages 1–18, 1996.

[Tod94] C.Todd and G.Davidson. Ac-3: Flexible perceptual coding for audio transmission and storage. *96th Convention of the Audio Engineering Society*, pages 89–102. AES, 1994.

[Tri99] Trimaran 2.0: An Infrastructure for Research in Instruction-Level Parallelism. http://www.trimaran.org, 1999.

[Tri02] K.Trivedi. Probability and Statistics with Reliability, Queuing and Computer Science Applications. John Wiley and Sons, New York, USA, 2002.

[Tri08] Trimaran 4.0: An Infrastructure for Research in Backend Compilation and Architecture Exploration. http://www.trimaran.org/docs/trimaran4_manual.pdf, 2008.

[Tul95] D.Tullsen, S.Eggers, and H.Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *Proc. of Intnl. Symposium on Computer Architecture (ISCA)*, pages 392–403, June 1995.

[Tys01] G.Tyso, M.Smelyanskiy, and E.Davidson. Evaluating the use of register queues in software pipelined loops. *IEEE Trans. on Computers*, pages 769–783, August 2001.

[UCB07] University of California Berkeley, http://bee2.eecs.berkeley.edu/. BEE2, 2007.

[Uh99] Gang-Ryung Uh, Yuhong Wang, David Whalley, Sanjay Jinturkar, Chris Burns, and Vincent Cao. Effective exploitation of a zero overhead loop buffer. *LCTES '99: Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pages 10–19, New York, NY, USA, 1999. ACM Press.

[Uni05] UNISIM: UNIted SIMulation environment. http://unisim.org/site/, 2005.

[Vda03] T.Vander Aa, M.Jayapala, F.Barat, G.Deconinck, R.Lauwereins, H.Corporaal, and F.Catthoor. Instruction buffering exploration for low energy embedded processors. *Proc. of 13th Intnl. Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS)*, September 2003.

[Vda04a] T.Vander Aa, M.Jayapala, F.Barat, G.Deconinck, R.Lauwereins, F.Catthoor, and H.Corporaal. Instruction buffering exploration for low energy vliws with instruction clusters. *Proc. of the Asian Pacific Design and Automation Conf. 2004 (ASPDAC'2004)*, Yokohama, Japan, January 2004.

[Vda04b] T.Vander Aa, M.Jayapala, F.Barat, G.Deconinck, R.Lauwereins, H.Corporaal, and F.Catthoor. Instruction buffering exploration for low energy embedded processors. *Journal of Embedded Computing*, 1(3), 2004.

[Vda05] T.Van der Aa. Low Energy Instruction Memory Exploration. PhD thesis, KULeuven, ESAT/ELECTA, 2005.

[VdW05] J.van de Waerdt, S.Vassiliadis, S.Das, S.Mirolo, C.Yen, B.Zhong, C.Basto, J.van Itegem, D.Amirtharaj, K.Kalra, P.odriguez, and H.van Antwerpen. The tm3270 media-processor. *MICRO '05: Proceedings of the 38th Annual IEEE/ACM Intnl. Symposium on Microarchitecture (MICRO'05)*, pages 331–342, Washington, DC, USA, 2005. IEEE Computer Society.

[Ven03] G.Venkataramani, W.Najjar, F.Kurdahi, N.Bagherzadeh, W.Bohm, and J.Hammes. Automatic compilation to a coarse-grained reconfigurable system-opn-chip. *Trans. on Embedded Computing Sys.*, 2(4):560–589, 2003.

[Ver98] I.Verbauwhede and M.Touriguian. A low power dsp engine for wireless communications. *J. VLSI Signal Process. Syst.*, 18(2):177–186, 1998.

[Ver04] M.Verma, L.Wehmeyer, and P.Marwedel. Dynamic overlay of scratch-pad memory for energy minimization. *Proc. of CODES*, pages 104–109, 2004.

[Ver07] M.Verma, P.Marwedel, "Advanced Memory Optimization Techniques for Low-Power Embedded Processors", ISBN 978-1-4020-5896-7, Springer Netherlands, 2007.

[Vij03] N.Vijaykrishnan, M.Kandemir, M.Irwin, H.Kim, W.Ye, and D.Duarte. Evaluating integrated hardware-software optimizations using a unified energy estimation framework. *IEEE Trans. on Computers*, 52(1):59–76, January 2003.

[Vir] Virage Logic, http://www.viragelogic.com/render/content.asp?pageid=667. Virage Logic SiWare Memory.

[VPr94] J.Van Praet, G.Goosens, D.Lanneer, H.De Man, Instruction set definition and instruction selection for asips. *Proc. 7th ACM/IEEE Intnl. Symp. on High-Level Synthesis*, Niagara-on-the-Lake, Canada, May 1994.

[Vol59] J.Volder, "The CORDIC trigonometric computing technique", *IRE Trans. on Electron. Computing*, Vol.EC-8, pp.330–334, Sept. 1959.

[Wan07] H.Wang, M.Miranda, F.Catthoor, and W.Dehaene. Synthesis of runtime switchable pareto buffers offering full range fine grained energy/delay trade-offs. *J. of Signal Processing Systems*, Springer, Nov. 2007.

[Wan09] H.Wang, M.Miranda, W.Dehaene, and F.Catthoor. Design and synthesis of pareto buffers offering large range run-time energy/delay trade-off via combined buffer size and supply voltage tuning. *IEEE Trans. on VLSI Systems*, Vol.17, pages 117–127, Jan. 2009.

[Wid95] B.Widrow, I.Kollar, M.Liu Statistical theory of quantization. *IEEE Trans. on Instrumentation and Measuremnt*, Vol.45, No.6, pp.353–361, 1995.

[Wie01] O.Wiess, M.Gansen, and T.Noll. A flexible datapath generator for physical oriented design. *Proc. of ESSCIRC*, pages 408–411, Sep 2001.

[Wie02] P.Wielage and K.Goossens. Networks on silicon: Blessing or nightmare? *Euromicro Symposium On Digital System Design*, 2002.

[Wie03] T.Wiegand, G.Sullivan, G.Bjontegaard, and A.Luthra. Overview of the H.264/AVC video coding standard. *IEEE Trans. on Circuits and Systems for Video Technology*, 13(7):560–576, July 2003.

[Wil95] P.Wilson, M.Johnstone, M.Neely, and D.Boles. Dynamic storage allocation: A survey and critical review. *1995 Intnl. Workshop on Memory Management*, pages 1–116. Springer-Verlag, 1995.

[Woh08] M.Woh, Y.Lin, S.Seo, S.Mahlke, T.Mudge, C.Chakrabarti, R.Bruce, D.Kershaw, A.Reid, M.Wilder, K.Flautner, "From SODA to scotch: The evolution of a wireless baseband processor", *Proc. of Intnl. Symp. on Microarchitecture (MICRO-41)*, pp.152–163, Nov. 2008.

[Wu95] Y.Wu. Strength reduction of multiplications by integer constants. *SIGPLAN Not.*, 30(2):42–48, 1995.

[Xtensa] Tensilica Xtensa processor documentation, http://www.tensilica.com/

[Xu04] J.Xu, W.Wolf, J.Henkel, and S.Chakradhar. A case study in power optimization of networks-on-chip. *Proc. of DAC*, 2004.

[Xue07] L.Xue, O.Ozturk, and M.Kandemir. A memory-conscious code parallelization scheme. *Proc. of the 44th annual conference on design automation*, pages 230–233, New York, NY, USA, 2007. ACM Press.

[Ye00] W.Ye, N.Vijaykrishnan, M.Kandemir, and M.Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. *Proc. of Design Automation Conf.*, pages 340–345, 2000.

[Ye02] T.Ye, L.Benini, and G.De Micheli. Analysis of power consumption on switch fabrics in network routers. *Proc. of DAC*, 2002.

[Yu04] P.Yu and T.Mitra. Scalable instructions identification for instruction-set extensible processors. *Proc. of CASES*, September 2004.

[Zal00a] J.Zalamea, J.Llosa, E.Ayguade, and M.Valero. Two-level hierarchical register file organization for vliw processors. *Microarchitecture, 2000. MICRO-33. Proceedings. 33rd Annual IEEE/ACM Intnl. Symposium on*, pages 137–146, 2000.

[Zha02] Y.Zhang and D.Chen. Efficient global register allocation for minimizing energy consumption. *SIGPLAN Not.*, 37(4):42–53, 2002.

[Zuc98] D.Zucker, R.Lee, and M.Flynn. An automated method for software controlled cache prefetching. *HICSS '98: Proceedings of the Thirty-First Annual Hawaii Intnl. Conf. on System Sciences-Volume 7*, page 106, Washington, DC, USA, 1998. IEEE Computer Society.

[Zyu98] V.Zyuban and P.Kogge. The energy complexity of register files. *Intnl. Symposium on Low-Power Electronics and Design*, pages 305–310, 1998.

[Zyu01] V.Zyuban and P.Kogge. Inherently lower-power high-performance superscalar architectures. *IEEE Trans. on Computers*, 50(3):268–285, March 2001.