Preeti Ranjan Panda
Aviral Shrivastava
B.V.N. Silpa
Krishnaiah Gummidipudi

Power-efficient System Design

Springer

Power-efficient System Design

Preeti Ranjan Panda • Aviral Shrivastava
B.V.N. Silpa • Krishnaiah Gummidipudi

Power-efficient System Design

Springer

Dr. Preeti Ranjan Panda
Department Computer Science
and Engineering
Indian Institute of Technology
Hauz Khas, 110016
New Delhi
India
panda@cse.iitd.ac.in

Aviral Shrivastava
Department of Computer Science
and Engineering
Arizona State University
699 South Mill Avenue
85281, Tempe
USA
Aviral.Shrivastava@asu.edu

B.V.N. Silpa
Department of Computer Science
and Engineering
Indian Institute of Technology
Hauz Khas, 110016
New Delhi
India
silpa@cse.iitd.ac.in

Krishnaiah Gummidipudi
Department of Computer Science
and Engineering
Indian Institute of Technology
Hauz Khas, 110016
India
krishna@cse.iitd.ac.in

ISBN 978-1-4419-6387-1 e-ISBN 978-1-4419-6388-8
DOI 10.1007/978-1-4419-6388-8
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2010929487

Printed on acid-free paper

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

Preface

The Information and communication technology (ICT) industry is said to account for 2% of the worldwide carbon emissions – a fraction that continues to grow with the relentless push for more and more sophisticated computing equipment, communications infrastructure, and mobile devices. While computers evolved in the direction of higher and higher performance for most of the latter half of the 20th century, the late 1990's and early 2000's saw a new emerging fundamental concern that has begun to shape our day-to-day thinking in system design – power dissipation. As we elaborate in Chapter 1, a variety of factors colluded to raise power-efficiency as a first class design concern in the designer's mind, with profound consequences all over the field: semiconductor process design, circuit design, design automation tools, system and application software, all the way to large data centers.

Power-efficient System Design originated from a desire to capture and highlight the exciting developments in the rapidly evolving field of power and energy optimization in electronic and computer based systems. Tremendous progress has been made in the last two decades, and the topic continues to be a fascinating research area. To develop a clearer focus, we have concentrated on the relatively higher level of design abstraction that is loosely called the *system level*. In addition to the extensive coverage of traditional power reduction targets such as CPU and memory, the book is distinguished by detailed coverage of relatively modern power optimization ideas focussing on components such as compilers, operating systems, servers, data centers, and graphics processors.

The book is primarily intended to serve as a graduate-level text. An elementary familiarity with digital design, computer architecture, and system software utilities such as compilers and operating systems, is necessary in order to appreciate the contents of the book. However, in Chapter 2, we do attempt to quickly cover some of the background material on which the later discussion is based. After reading this book, the reader can expect to obtain a reasonable understanding of the opportunities for introducing power and energy efficiency into electronic and computer based systems, the current approaches targeting this objective, and the challenges ahead.

We gratefully acknowledge the feedback given by Sorav Bansal, Kolin Paul, Neeraj Goel, Anoop Nair, and Kumar S. S. Vemuri on an earlier version of the manuscript. We owe our gratitude to Chuck Glaser for his words of encouragement and valuable advice during the writing process; it has been an immense pleasure working with him.

Finally, we would like to place on record our heartfelt thanks to Lima Das for working very hard to design the impressive and very appropriately conceived cover artwork.

New Delhi, India and Tempe, USA
June 2010

Preeti Ranjan Panda
Aviral Shrivastava
B.V.N. Silpa
Krishnaiah Gummidipudi

Contents

Chapter 1
Low Power Design: An Introduction

1.1 The Emergence of Power as an Important Design Metric

Through most of the evolution of the electronics and computer industry in the twentieth century, technological progress was defined in terms of the inexorable march of *density* and *speed*. Increasing density resulted from process improvements that led to smaller and smaller geometries of semiconductor devices, so the number of transistors packed into a given area kept increasing. Increasing processing speed indirectly resulted from the same causes: more processing power on the same chip meant more computation resources and more data storage on-chip, leading to higher levels of parallelism and hence, more computations completed per unit time. Circuit and architectural innovations were also responsible for the increased speed. In the case of processors, deeper pipelines and expanding parallelism available led to improvements in the effective execution speed. In the case of memories, density continued to make remarkable improvements; performance improvements were not commensurate, but nevertheless, architectural innovations led to increased bandwidth at the memory interface.

In parallel to the steady march in the density and performance of high-end chips such as processors, a quiet mobile revolution was ignited in the 1990s. The specification for components targeting mobile systems were very different from the high-end processors. These chips had to fit into very tight power budgets defined by battery life considerations, which led to a significant investment in low power design techniques, methodologies, and tools. Starting early 1990s, researchers began to consider modifying the entire electronic design automation tool chain by incorporating power awareness. Even as the complexity of Systems-on-a-Chip (SoC) increased significantly and proceeded in the direction of integration of a large number of modules (cores), the individual cores were still designed to tight power constraints and intelligence was introduced in the overall power management of these SoCs in order to meet chip-level power budgets. Many of these cores were processors in themselves, but these were architected very differently from the standard high-end desktop or server processor – speed of individual cores was heavily compromised to meet power constraints by keeping the design simple. It was a prescient move – the high-end multicore processors of the early 21st century would later adopt the same philosophy.

P.R. Panda et al., *Power-efficient System Design*, DOI 10.1007/978-1-4419-6388-8_1,

Through the late 1990s and early 2000s, the parallel evolution of the two seemingly unrelated lines of products continued unabated. High end processor systems delivered on the traditional density and performance scaling promise through increased architectural sophistication, while the mobile revolution was well under way, spearheaded by the ubiquitous cell phone.

Towards the mid-2000s, important directional changes were visible in the evolution of these technologies. Cell phone makers began packing more and more functionality into their devices, making them look increasingly similar to general purpose programmable devices. All the functionality still had to be delivered within the tight power envelope of a few watts – imposed by the simple physical requirement that the device should fit and stay within the pocket without burning a hole through it. Desktop processors, at the same time, ran up against an unexpected barrier – the *power wall*, which imposed an upper limit on the power dissipation. It turned out that, if the processor power exceeded around 150 W, then the simple fan-based cooling mechanism used in desktop computers might be inadequate. More sophisticated cooling would increase the cost of the systems to inordinate levels – the consumer had got used to price drops with increasing performance over the years. Suddenly, power related innovation became a top concern even in high-end processors, altering the designer's priorities radically after decades of riding the predictable performance and density curves.

A third front had opened with the exploding Internet revolution – the world was getting more and more connected, powered by infrastructural re-organization that was not obviously visible to the end-user, but nevertheless, an important enabling innovation. Large scale distributed storage and processing capabilities had become necessary. To handle the terabytes of data continuously being generated, stored, and retrieved, data had to be organized in non-trivial ways. Similarly, the huge processing and computation power also needed similar architectural paradigm shifts. *Data centers* were the offshoot of these pressing requirements. Designers realized the economies of scale provided by aggregations of tens of thousands of computers stacked into a warehouse-style environment, which would make more efficient utilization of space than the maintenance of a large number of small server rooms in individual companies. Service models began to evolve based on this idea. There was just one issue – power. Large data centers draw megawatts of power. It is well established that the annual energy cost of operating a large data center rivals the cost of the equipment itself. The time had come to start worrying about ways to reduce the energy bills. Further, the concentration of a large number of powerful computers in a data center leads to a very high thermal density, with significant consequences on the cooling technology required to maintain the systems within the operating temperature range, ultimately impacting the cost of maintaining the infrastructure.

As we proceed to the 2010s, it is amply clear that no one can run away from the looming power wall. Far from being limited to battery-operated devices, power and energy efficiency now has to be built into products at every level of abstraction, starting from the semiconductor device level, through circuit and chip levels, all the way to compilers, operating systems, and large data centers. While earlier generations of hardware designers invariably optimized the product for high performance and reported the power dissipation merely as a documentation exercise to generate complete data sheets for the customer, newer generations of designers now need to aggressively optimize for power – even at the expense of performance. Power is now elevated to what is referred to as a *first class design concern*.

Power awareness on the software side of the computation stack expressed itself as an issue somewhat later, as expected. This followed after appropriate hooks were provided by the hardware. It didn't take long for researchers to realize that a huge opportunity lay ahead for power aware system and application software. This followed the observation that at least some of the hardware power optimizations would work better if there were hints passed on to the hardware by the software and vice versa. For example, a higher level macroscopic view of a system as a whole is available to an application designer or a compiler that is able to view and analyze an extended piece of code, than a processor that is able to view only one small window of instructions at a time. Power efficiency demands are now placed on compilers, application developers, and operating systems.

The need for energy efficiency in electronic and computer systems tracks the global need for energy efficiency in every other walk of life. At a time when the world is acutely aware of the environmental consequences of the huge collective footprint of our individual actions, and struggling to evolve a sustainable solution, it is natural that power efficiency and energy efficiency concerns should permeate into everything we do. With electronic and computer systems beginning to play such a dominant role in our day-to-day lives, they present themselves as perfect targets on which to focus our power efficiency efforts.

1.2 Power Efficiency vs. Energy Efficiency

Since the terms *power efficiency* and *energy efficiency* are both used in literature, often interchangeably, let us look at the elementary definitions of power and energy and clarify the distinction between power and energy efficiency and between the objectives of power and energy optimizations.

Figure 1.1 shows the instantaneous power dissipated by a system S as a function of time. The power $P(t)$ will vary from time to time because the work done inside the system typically changes with time. In electronic systems, power is, in turn, determined by the current drawn by the system and the voltage at which it operates, as a function of time:

$$P(t) = V(t) \times I(t) \tag{1.1}$$

where V and I are the instantaneous voltage and current respectively. If the external supply voltage remains constant, as is common in many systems, power is directly proportional to the current drawn, and the power curve essentially tracks the variation of the current over time.

What about energy dissipation by the same system S? Power is simply the rate at which energy is dissipated, so the energy dissipated over a period $t = 0$ to $t = T$ is given by:

$$E = \int_0^T P(t)\, dt \tag{1.2}$$

This corresponds to the area under the $P(t)$ curve as indicated in Fig. 1.1.

A system S' that delivers the same work as the one in Fig. 1.1 in the same duration T, but is characterized by a different power curve $P'(t)$, as shown in Fig. 1.2(a), would be called more power-efficient because the P' curve is always lower than P. What about its energy dissipation? Its energy E' corresponds to the area under the P' curve, which is clearly less than E. Hence, E' is also more energy-efficient.

On the other hand, let us consider the power characteristic of a different system S'' shown in Fig. 1.2(b). System S'' has a power curve P'' that is under P most of

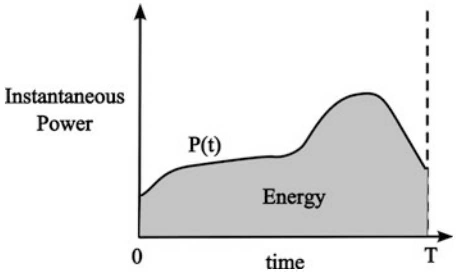


Fig. 1.1 The relationship between power and energy: the area under the power curve over a time interval gives the energy consumed

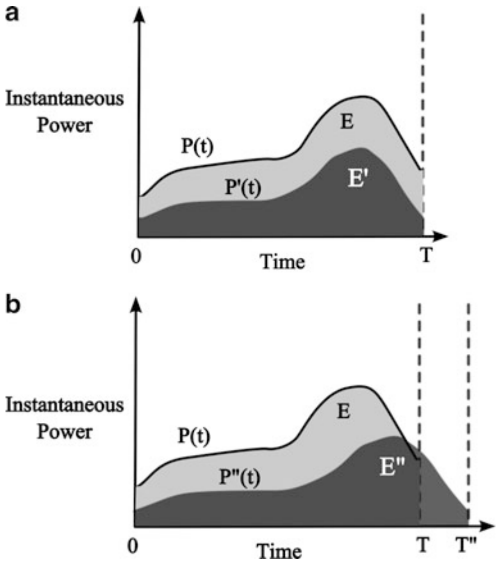


Fig. 1.2 Power efficiency and energy efficiency. Three systems S, S', and S'' have power curves P, P', and P'' respectively. (**a**) S' is more power-efficient and more energy-efficient than S. (**b**) S'' has lower peak power than S. S'' has lower average power/energy than S in time period $[0, T'']$ if $E'' < E$

the time, but it takes longer to complete its task ($T'' > T$). The energy dissipated is now given by:

$$E'' = \int_0^{T''} P''(t)\,dt \tag{1.3}$$

Which of S, S'' is more power efficient? Which is more energy-efficient?

The energy efficiency question is easier to answer – S'' is more energy efficient if $E'' < E$. This is a clearer question because energy corresponds to the total work done by a system, independent of completion time.

Power, on the other hand, is intricately tied to completion times, and the definition of power-efficiency needs to be clarified in a given context. Two major power efficiency criteria are commonly relevant:

1. Peak power
2. Average power

The peak power criterion is sometimes externally imposed on a system. If minimizing peak power is the objective, then S'' is more efficient. If a peak power constraint is imposed on the system, then the design has to be adjusted to ensure that the power curve is always under the horizontal line corresponding to the power constraint, irrespective of how long it takes for completion.

Average power is defined with respect to a time interval. In Fig. 1.2(b), if we take the time interval $[0, T'']$, then we have:

$$\text{Average Power for } \; S = \frac{E}{T''} \tag{1.4}$$

$$\text{Average Power for } \; S'' = \frac{E''}{T''} \tag{1.5}$$

Note that, for a fair comparison of the two systems, the time durations should be the same. S' has completed its task at time T, and is consuming zero power between T and T'', which should be accounted for in the average power calculation. Since the two denominators are the same, the relationship between the average power dissipation between the two systems is identical to the relationship between the total energy.

Requirements for energy efficiency or average power reduction usually stem from battery life considerations. To a level of approximation, battery capacity can be viewed in terms of a fixed amount of total energy that it can supply.

Requirements for peak power usually stem from two sources:

- external power supply may impose a constraint on the maximum instantaneous power that can be supplied; or
- there may be a cap on the maximum temperature (which is indirectly determined by peak power) that a device is allowed to reach. This could be applicable to both mobile devices as well as powerful server computers.

In this book we discuss different classes of mechanisms that target the reduction of either energy (average power) or peak power. The actual objective is clarified from the context in which it appears.

1.3 Power-Performance Tradeoff

Figure 1.2(b) illustrates an important trade-off between peak power and performance that is not merely theoretical. The trade-off is real and has tremendously influenced the direction taken by the mobile computer industry. The explosion of interest in ultra-portable computers (also referred to as *netbooks*) in recent times, along with the evolution of cell phones into more sophisticated gadgets that make them look more like handheld computers, clearly shows that there is a huge potential for the market segment where computer systems are held within a tight power budget while sacrificing performance. Netbooks run on processors with very low peak power and correspondingly weak performance; their acceptance in the market may imply that the average set of applications run by many users just don't require very high speed. Instead of raw speed, other important features such as network connectivity, mobility, and battery life are valued. The race to deliver higher and higher raw performance may have transformed into a more nuanced battle, with new battle grounds being created that squarely put power and energy at the center.

Design metrics such as *energy-delay product* are some times used to evaluate the relative merits of two implementations in a way that accounts for both energy and performance [1]. Such metrics help in discriminating between alternatives that are equivalent in total energy dissipation but different in performance. Energy-delay products for the two systems above are given by:

$$\text{Energy-Delay Product for } S = E \times T$$
$$\text{Energy-Delay Product for } S'' = E'' \times T''$$

The term *performance-per-watt* is sometimes used in the industry to characterize energy efficiency of hardware [2]. This term is effectively a direct or indirect computation of $1/E$, with E defined as above. The *performance* measure here can be interpreted in different ways, for example, execution frequency or throughput of computation. If performance is interpreted as the inverse of latency T, then we have:

$$\text{Performance-per-watt} = \frac{1}{T} \times \frac{1}{P} = \frac{1}{E}$$

Higher performance-per-watt means more energy-efficient (lower total energy) and vice versa.

The performance-per-watt metric gives equal emphasis to power and performance, while the energy-delay product ($E \times T = P \times T^2$) gives higher emphasis to performance (the latter varies quadratically with delay).

1.4 Power Density

Power density is an additional important concern that is not captured in a straightforward way by the power and energy efficiency metrics discussed above. This refers to the fact that the distribution of power dissipation across a chip is never uniform because different modules exhibit different degrees of activity in any fixed period of time. Thus, the power density – the dynamic energy dissipated per unit area – is non-uniformly distributed on the chip, leading to *hotspots* where the power density is significantly high. Higher power density leads to higher temperatures, and a non-uniform power density leads to a non-uniform temperature variation on the chip, leading to some associated problems. Even if the average and peak power numbers are acceptable, power density variations can cause regions on the chip to have unacceptably high temperatures leading to failures. Thus, the chip designer also needs to target a relatively uniform distribution of power density in the resulting chip.

1.5 Power and Energy Benchmarks

The rising importance of power and energy motivated the eventual development of benchmarking efforts targeting the evaluation of different classes of systems with respect to power and energy. *Powerstone* [3] was among the early reported efforts targeting power evaluation of mobile embedded systems. The Powerstone benchmark contained applications selected from the following domains: automobile control (*auto* and *engine*), signal and image processing (*fir_int* – integer FIR filter, *jpeg* – JPEG decompression, *g721* – voice compression, and *g3fax* – fax decode), graphics (*bilt*), and several utilities (*compress* – UNIX file compression, *crc* – cyclic redundancy check, *bilv* – bit vector operations, *des* – data encryption, etc.).

Joulesort [4] is a more modern benchmarking effort aimed at power evaluation of high-end server, cluster, and data center class computer systems. The benchmark consists of essentially an application for sorting a randomly permuted collection of records. Different categories of the benchmark contain different input data sizes (10 GB, 100 GB, and 1 TB). Thus, the benchmark has I/O as the primary emphasis, with secondary emphasis on CPU and memory. Joulesort is good at exercising whole system level power efficiencies. The proposed evaluation metric used here is the energy consumed for processing a fixed input size.

1.6 Power Optimizations at the System Level

In this book we will explore power optimizations at the *system level* of abstraction. Since the word *system* is very generic, it is important to clarify its interpretation as used in this book. The following different levels of abstraction could possibly be considered in the context of power optimizations. They are arranged in the increasing level of granularity.

Device level. This refers to the choice of the appropriate semiconductor materials and processes used to fabricate transistors and other devices in this class which form the building blocks of electronic technology.

Circuit Level. This refers to the interconnection of transistors and the related class of components, along with the choice of appropriate geometry for these devices.

Gate Level. This refers to the logic level of abstraction, where the components (*gates*) implement simple combinational and sequential functions (such as AND, MUX, flip-flops, etc.). A logic synthesis tool generates an optimized logic netlist starting from an unoptimized gate level description, or some other simple format such as a truth table.

Register Transfer Level (RTL). This refers to a description that is usually expressed in a Hardware Description Language (HDL), containing a cycle-accurate description of the hardware to be designed. RTL synthesis tools automate the generation of a gate netlist from an RTL HDL description. The architecture view of a processor or other hardware falls in this level of abstraction.

Behavioral Level. At this level, the hardware is still described in an HDL, but the level of detail is lesser than RTL. For example, we may specify an algorithm implemented in hardware, but indicate nothing about which operation is performed in which clock cycle. A behavioral synthesis tool takes a behavioral level model as input, performs a series of *high level synthesis* optimizations, and finally generates an RTL HDL design, to be processed further.

Transaction Level. At this level – pertaining to SoC design – the design is no longer a pure hardware design. In fact, the decision of which part of the specification will go into hardware and which into software, has not been finalized yet. The details of the computation in individual blocks and communication across blocks of the SoC are not specified yet.

Application and System Software Level. At this level, we have moved to software executing on a processor hardware. This level encompasses all application software and key system software such as compilers that have a deeper view of a single application, and operating systems that have a wider view of the run-time environment.

Full System, Server, and Data Center Level. This level of abstraction refers to an aggregation of electronic and computer systems that can be considered complete in some sense - for example, cell phones, laptops, dedicated servers, all the way up to data centers.

Our focus on system-level design covers the behavioral and higher levels. That is, in this book we focus on the Behavioral Level, Transaction Level, Application and System Software Level, and Full System, Server, and Data Center Levels. Key power optimization strategies at lower levels of abstraction are summarized briefly in Chapter 2.

1.7 Organization of this Book

In this book we explore various mechanisms that have been proposed for reducing power dissipation in electronic and computer systems. Figure 1.3 gives a pictorial view of the contents of the book. The organization of the chapters is as follows.

In Chapter 2 ("Basic Low Power Digital Design") we introduce the avenues for power dissipation in electronic systems based on CMOS processes, and the basic techniques employed for reducing dynamic and static power. The chapter covers

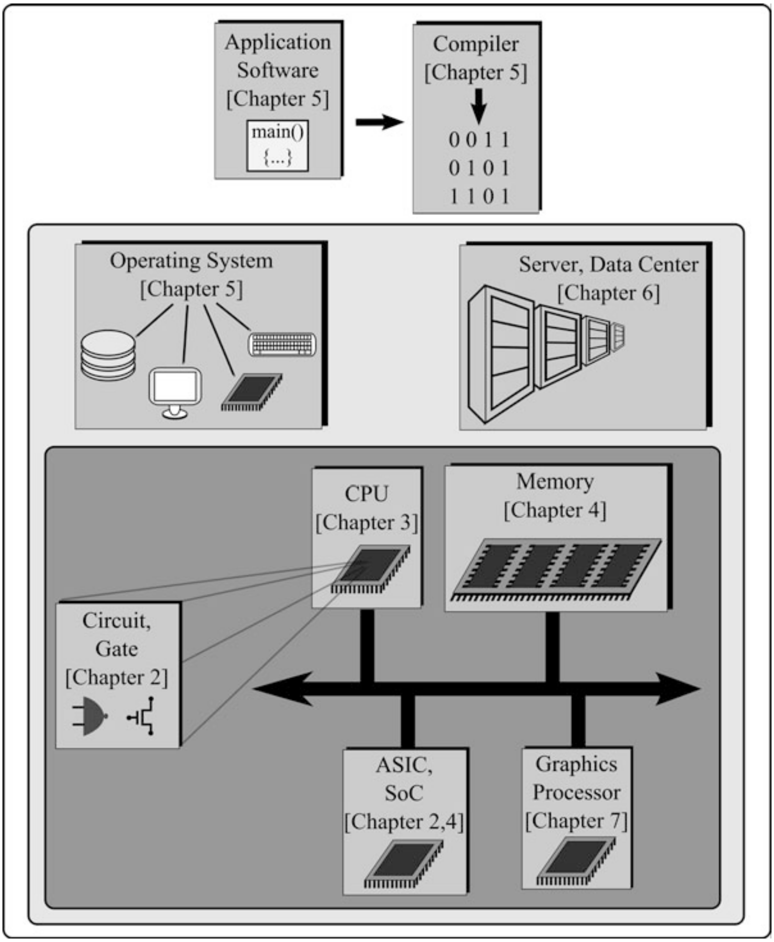


Fig. 1.3 Contents and organization of this book. Chapter 2 outlines the electronic foundations and basic synthesis/automation techniques. Chapter 3 and 4 focus on CPU and memory respectively. Chapter 5 discusses power-aware operating systems, compilers, and applications. Chapter 6 covers servers and data centers. Chapter 7 describes low power graphics processors

the electronic foundations that form the basis of most of the higher level power optimizations discussed in subsequent chapters.

In Chapter 3 ("Power-efficient Processor Architecture") we delve into the details of modern processor architecture and point out some of the major strategies for reducing power in various components. While circuit level enhancements for power reduction naturally apply to processor design, our focus here is on the architecture-level modifications that reduce power.

In Chapter 4 ("Power-efficient Memory and Cache") we discuss the basics of memory architecture and point out several techniques for reducing memory power in different contexts. In addition to power-aware cache memory design in the context of processors, we also discuss power reduction opportunities afforded by scratch pad memory and banked memory in the SoC context, as well as other components such as translation look-aside buffers and dynamic memory (DRAM).

In Chapter 5 ("Power Aware Operating Systems, Compilers, and Application Software") we move up another level in the compute stack – system software such as operating systems and compilers, and application software. Once the proper power optimization hooks are provided by the hardware, new opportunities open up with regard to their exploitation in software, starting from power-aware compilers.

In Chapter 6 ("Power Issues in Servers and Data Centers") we examine power awareness issues being introduced into high-end server-class computers and aggregations of servers constituting data centers. High level operating system features such as task allocation can now take power into account, exploiting mechanisms provided by the hardware for controlling performance and power states of individual computers.

In Chapter 7 ("Low Power Graphics Processors") we do a detailed study of system level power optimizations in graphics processors. The chapter gives a brief introduction to the graphics processing domain and proceeds to illustrate the application of several of the ideas introduced in previous chapters in the context of a complex processing system whose components bear some similarity to general purpose processors, but nevertheless, the overall architecture is application specific.

References

1. Gonzalez, R., Horowitz, M.: Energy dissipation in general purpose microprocessors. IEEE Journal of Solid-State Circuits **31**(9), 1277–1284 (1996)
2. Laudon, J.: Performance/watt: the new server focus. SIGARCH Computer Architecture News **33**(4), 5–13 (2005)
3. Malik, A., Moyer, B., Cermak, D.: The M·CORE$^{\mathrm{TM}}$ M340 unified cache architecture. In: Proceedings of the IEEE International Conference On Computer Design: VLSI in computers & processors, pp. 577–580 (2000)
4. Rivoire, S., Shah, M.A., Ranganathan, P., Kozyrakis, C.: Joulesort: a balanced energy-efficiency benchmark. In: SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pp. 365–376. ACM, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1247480.1247522

Chapter 2
Basic Low Power Digital Design

Moore's law [12], which states that the "number of transistors that can be placed inexpensively on an integrated circuit will double approximately every two years," has often been subject to the following criticism: while it boldly states the blessing of technology scaling, it fails to expose its bane. A direct consequence of Moore's law is that the "power density of the integrated circuit increases exponentially with every technology generation". History is witness to the fact that this was not a benign outcome. This implicit trend has arguably brought about some of the most important changes in electronic and computer designs. Since the 1970s, most popular electronics manufacturing technologies used bipolar and nMOS transistors. However, bipolar and nMOS transistors consume energy even in a stable combinatorial state, and consequently, by 1980s, the power density of bipolar designs was considered too high to be sustainable. IBM and Cray started developing liquid, and nitrogen cooling solutions for high-performance computing systems [5,11,16,19,21,23–25]. The 1990s saw an inevitable switch to a slower, but lower-power CMOS technology (Fig. 2.1). CMOS transistors consume lower power largely because, to a first order of approximation, power is dissipated only when they switch states, and not when the state is steady. Now, in the late 2000s, we are witnessing a paradigm shift in computing: the shift to multi-core computing. The power density has once again increased so much that there is little option but to keep the hardware simple, and transfer complexity to higher layers of the system design abstraction, including software layers.

This chapter explains the basics of power consumption and dissipation in the operation of CMOS transistors, and also discusses some of the fundamental mechanisms employed for power reduction.

2.1 CMOS Transistor Power Consumption

The power consumption of a CMOS transistor can be divided into three different components: dynamic, static (or leakage) and short circuit power consumption. Figure 2.2 illustrates the three components of power consumption. Switching power,

P.R. Panda et al., *Power-efficient System Design*, DOI 10.1007/978-1-4419-6388-8_2,

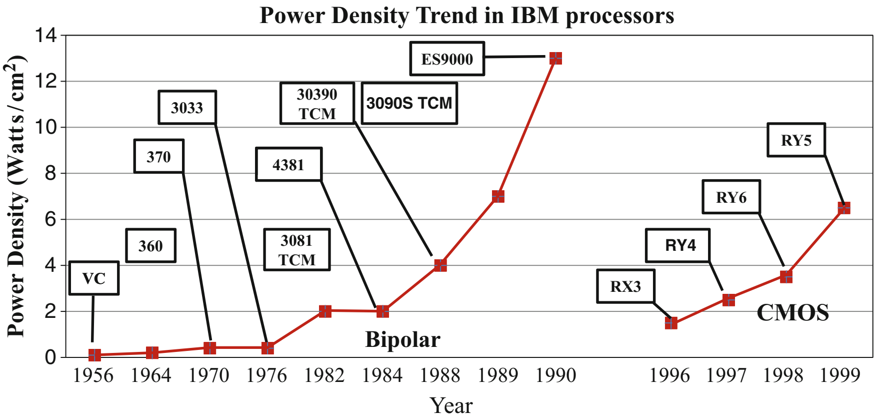
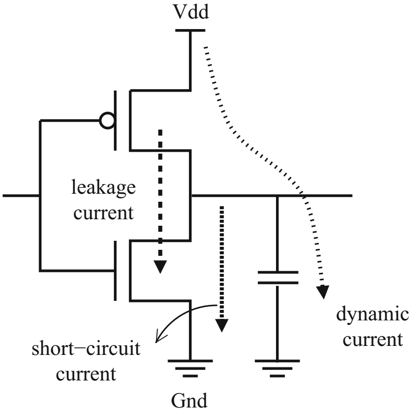


Fig. 2.1 Unsustainable increase in the power density caused a switch from fast, but high-power bipolar transistors to slow, but low-power CMOS transistor technology. We are at the doorsteps of the same problem again (Data courtesy IBM Corp)

Fig. 2.2 The dynamic, short circuit and leakage power components of transistor power consumption. Dynamic and short circuit power are also collectively known as switching power, and are consumed when transistors change their logic state, but leakage power is consumed merely because the circuit is "powered-on"



which includes both dynamic power and short-circuit power, is consumed when signals through CMOS circuits change their logic state, resulting in the charging and discharging of load capacitors. Leakage power is primarily due to the sub-threshold currents and reverse biased diodes in a CMOS transistor. Thus,

$$P_{total} = P_{dynamic} + P_{short-circuit} + P_{leakage}. \tag{2.1}$$

2.1.1 Switching Power

When signals change their logic state in a CMOS transistor, energy is drawn from the power supply to charge up the load capacitance from 0 to V_{dd}. For the inverter example in Fig. 2.2, the power drawn from the power supply is dissipated as heat in pMOS transistor during the charging process. Energy is needed whenever charge is moved against some potential. Thus, $dE = d(QV)$. When the output of the invertor transitions from logical 0 to 1, the load capacitance is charged. The energy drawn from the power supply during the charging process is given by,

$$dE_P = d(VQ) = V_{dd}.dQ_L$$

since the power supply provides power at a constant voltage V_{dd}. Now, since $Q_L = C_L.V_L$, we have:

$$dQ_L = C_L.dV_L$$

Therefore,

$$dE_P = V_{dd}.C_L.dV_L$$

Integrating for full charging of the load capacitance,

$$\begin{aligned} E_P &= \int_0^{V_{dd}} V_{dd}.C_L.dV_L \\ &= V_{dd}.C_L.\int_0^{V_{dd}} dV_L \\ &= C_L.V_{dd}^2 \end{aligned} \tag{2.2}$$

Thus a total of $C_L.V_{dd}^2$ energy is drawn from the power source. The energy E_L stored in the capacitor at the end of transition can be computed as follows:

$$dE_L = d(VQ) = V_L.dQ_L$$

where V_L is the instantaneous voltage across the load capacitance, and Q_L is the instantaneous charge of the load capacitance during the loading process. Therefore,

$$dE_L = V_L.C_L dV_L$$

Integrating for full charging of the load capacitance,

$$\begin{aligned} E_L &= \int_0^{V_{dd}} C_L.V_L.dV_L \\ &= \frac{(C_L.V_{dd}^2)}{2} \end{aligned} \tag{2.3}$$

Comparing Equations 2.2 and 2.3, we notice that only half of the energy drawn from the power supply is stored in the load capacitance; the rest is dissipated as heat. This $\frac{1}{2}\ C_L.V_{dd}^2$ energy stored in the output capacitance is released during the discharging of the load capacitance, which occurs when the output of the invertor transitions from logical 1 to 0. The load capacitance of the CMOS logic gate consists of the output node capacitance of the logic gate, the effective capacitance of the interconnects, and the input node capacitance of the driven gate.

2.1.2 Short Circuit Power

Another component of power, *short-circuit power* (also known as *crowbar power*, or —em rush-through power) becomes important because of finite non-zero rise and fall times of transistors, which causes a direct current path between the supply and ground. This power component is usually not significant in logic design, but it appears in transistors that are used to drive large capacitances, such as bus wires and especially off-chip circuitry. As wires on chip became narrower, long wires became more resistive. CMOS gates at the end of those resistive wires see slow input transitions. Consider the inverter in Fig. 2.2. Figure 2.3 shows the variation of the short circuit current I_{SC} as the inverter is driven by a rising ramp input from time 0 to T_R. As the input voltage rises, at time t_0, we have $V_{in} > VT_n$, i.e., the input voltage become higher than the threshold voltage of the nMOS transistor. At this time a short-circuit current path is established. This short circuit current increases as the nMOS transistor turns "on". Thereafter, the short circuit current first increases and then decreases until, after t_1, we have $V_{in} > V_{dd}-VT_p$, i.e., the pMOS transistor turns off, signalling the end of short-circuit current. Therefore, in the duration when

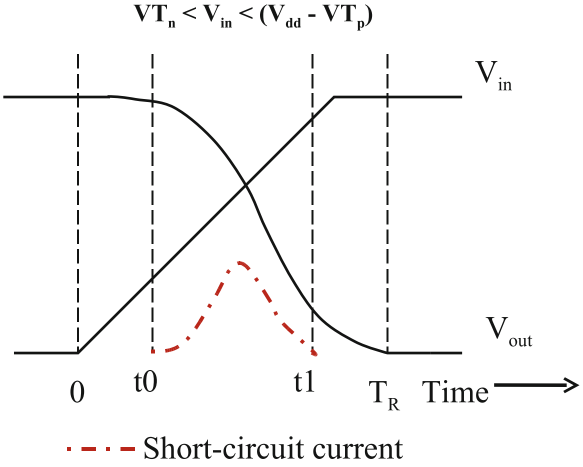


Fig. 2.3 Short circuit power is consumed in a circuit when both the nMOS and pMOS transistors are "on"

$(VT_n < V_{in} < (V_{dd} - VT_p))$ holds, there will be a conductive path open between V_{dd} and GND because both the nMOS and pMOS devices will be simultaneously on. Short-circuit power is typically estimated as:

$$P_{short-circuit} = 1/12.k.\tau.F_{clk}.(V_{dd} - 2V_t)^3 \tag{2.4}$$

where τ is the rise time and fall time (assumed equal) and k is the gain factor of the transistor [22].

2.1.3 Leakage Power

The third component of power dissipation in CMOS circuits, as shown in Equation 2.1, is the static or leakage power. Even though a transistor is in a stable logic state, just because it is powered-on, it continues to leak small amounts of power at almost all junctions due to various effects. Next we discuss about the significant components of leakage power.

2.1.3.1 Reverse Biased Diode Leakage

The reverse biased diode leakage is due to the reverse bias current in the parasitic diodes that are formed between the diffusion region of the transistor and substrate. It results from minority carrier diffusion and drift near the edge of depletion regions, and also from the generation of electron hole pairs in the depletion regions of reverse-bias junctions. As shown in Fig. 2.4, when the input of inverter in Fig. 2.2 is high, a reverse potential difference of V_{dd} is established between the drain and the n-well, which causes diode leakage through the drain junction. In addition, the n-well region of the pMOS transistor is also reverse biased with respect to the p-type substrate. This also leads to reverse bias leakage at the n-well junction. The reverse bias leakage current is typically expressed as,

$$I_{rbdl} = A.Js.(e^{q \cdot V_{bias}/kT} - 1) \tag{2.5}$$

where A is the area of the junction, J_s is the reverse saturation current density, and V_{bias} is the reverse bias voltage across the junction, and $V_{th} = kT/q$ is the thermal voltage. Reverse biased diode leakage will further become important as we continue to heavily dope the n- and p-regions. As a result, zener and band-to-band tunneling [14] will also become contributing factors to the reverse bias current.

2.1.3.2 Gate Induced Drain Leakage

Gate-induced drain leakage (GIDL) is caused by high field effect in the drain junction of MOS transistors [2]. In an nMOS transistor, when the gate is biased to form

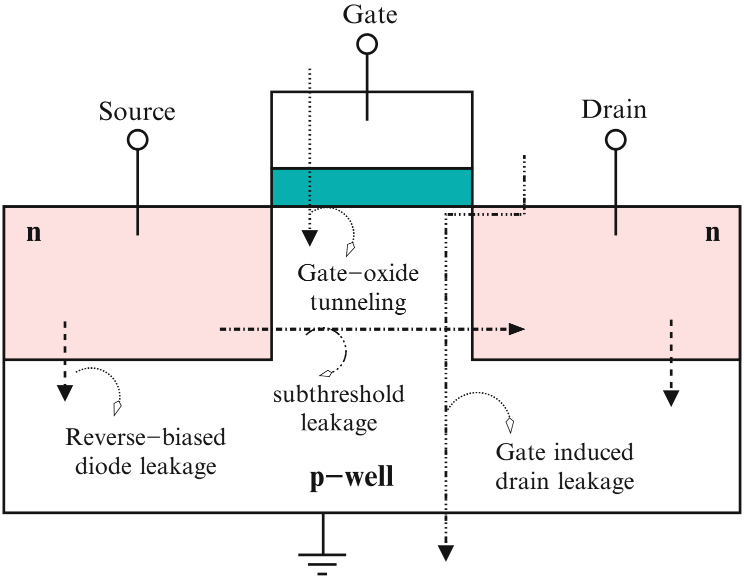


Fig. 2.4 Components of Leakage Power: (i) Subthreshold current flows between source and drain; (ii) Reverse-biased diode leakage flows across the parasitic diodes; (iii) Gate induced drain leakage flows between the drain and substrate

accumulation layer in the silicon surface under the gate, the silicon surface has almost the same potential as the p-type substrate, and the surface acts like a p-region more heavily doped than the substrate. However, when the gate is at zero or negative voltage and the drain is at the supply voltage level, there can be a dramatic increase of effects like avalanche multiplication and band-to-band tunneling. Minority carriers underneath the gate are swept to the substrate, creating GIDL current. GIDL current or I_{GIDL} is typically estimated as:

$$I_{GIDL} = AE_s.e^{-\frac{B}{E_s}} \tag{2.6}$$

where E_s is the transverse electric field at the surface [3]. Thinner oxide T_{ox} and higher supply voltage V_{dd} increase GIDL. GIDL is also referred to as surface band-to-band tunneling leakage.

2.1.3.3 Gate Oxide Tunneling

Gate oxide tunneling current I_{ox}, flows from the gate through the oxide insulation to the substrate. In oxide layers thicker than 3-4 nm, this type of current results from the Fowler-Nordheim tunneling of electrons into the conduction band of the oxide layer under a high applied electric field across the oxide layer [17]. As oxides get thinner, this current could surpass many other smaller leakages, e.g., weak inversion and DIBL as a dominant leakage mechanism in the future. I_{ox} is typically estimated as:

$$I_{ox} = A.E_{ox}^2.e^{-\frac{B}{E_{ox}}} \tag{2.7}$$

where E_{ox} is the electric field across the oxide.

In oxide layers less than 3-4 nm thick, there can also be direct tunneling through the silicon oxide layer. Mechanisms for direct tunneling include electron tunneling in the conduction band (ECB), electron tunneling in the valence band (EVB), and hole tunneling in the valence band (HVB).

2.1.3.4 Subthreshold Leakage

Subthreshold current flows from the source to drain even if the gate to source voltage is below the threshold voltage of the device. This happens due to several reasons. First is the weak inversion effect: when the gate voltage is below V_T, carriers move by diffusion along the surface similar to charge transport across the base of bipolar transistors. Weak inversion current becomes significant when the gate to source voltage is smaller than but very close to the threshold voltage of the device. The second prominent effect is the Drain-Induced Barrier Lowering (DIBL). DIBL is essentially the reduction of threshold voltage of the transistor at higher drain voltages. As the drain voltage is increased, the depletion region of the p-n junction between the drain and body increases in size and extends under the gate, so the drain assumes a greater portion of the burden of balancing depletion region charge, leaving a smaller burden for the gate. As a result, the charge present on the gate retains the charge balance by attracting more carriers into the channel, an effect equivalent to lowering the threshold voltage of the device. DIBL is enhanced at higher drain voltage and shorter effective channel length (L_{eff}) [14]. The third effect is the direct punch-through of the electrons between drain and source. It occurs when when the drain and source depletion regions approach each other and electrically "touch" deep in the channel. In a sense, punch-through current is a subsurface version of DIBL.

As a combination of all these sub-currents, I_{sub} is typically modeled as:

$$I_{sub} = I_0 e^{\frac{V_G - V_S - V_{T0} - \gamma V_S + \eta V_{DS}}{n V_{th}}} \left(1 - e^{\frac{-V_{DS}}{V_{th}}}\right), \tag{2.8}$$

where $V_{th} = kT/q$ is the thermal voltage, n is the subthreshold swing coefficient constant, γ is the linearized body effect coefficient, η is the DIBL coefficient, and I_0 is the technology dependent subthreshold leakage which can be represented as,

$$I_0 = \mu_0 C_{ox} \frac{W}{L} V_{th}^2 e^{1.8}. \tag{2.9}$$

2.2 Trends in Power Consumption

At the macro-level, the most significant trend is the increasing contribution of leakage power in the total power dissipation of an electronic system designed in CMOS technology. For a long time, the switching component of the dynamic power was the major component of the total power dissipated by a circuit. However, in order to keep power dissipation and power delivery costs under control, the operational voltage V_{dd} was scaled down at the historical rate of 30% per technology generation. In conjunction, to improve transistor and circuit performance the threshold voltage V_t was also reduced at the same rate so that a sufficiently large gate overdrive $(V_{dd} - V_t)$ is maintained. However, as seen from Equation 2.8, reduction in V_t causes transistor subthreshold leakage current (I_{sub}) to increase exponentially. Furthermore, other components of leakage current, e.g., the gate leakage and reverse-biased junction Band To Band Tunneling (BTBT) become important as we scale fabrication technology to 45 nm and downwards. Other factors such as gate-induced drain leakage (GIDL) and drain-induced barrier lowering (DIBL) will also become increasingly significant.

In addition to the increasing dominance of leakage power, the subthreshold leakage and the gate-oxide tunneling increase extremely rapidly (exponentially) with technology scaling, and dwarf dynamic power. We are already at a point where $V_{dd} - V_t$ is low, and leakage power is comparable to dynamic switching power, and in some cases, may actually even dominate the overall power dissipation. Large leakage comes with several associated problems such as lower noise immunity of dynamic logic circuits, instability of SRAM cells, and eventually, lower yield.

Another dimension of worry is added by the fact that unlike dynamic power, leakage power increases exponentially with temperature. In order to improve performance we have been continuously scaling the supply and threshold voltages. While this results in high frequency of operation, temperatures rise due to large active power consumption. The high temperature increases the sub-threshold leakage (which is a strong function of temperature), further increasing temperature. This circular situation is depicted in Fig. 2.5. If heat cannot be dissipated effectively, a positive feedback between leakage power and temperature can result in thermal runaway. Such a situation can have disastrous consequences, including permanent physical damage of the circuit. Most processors are now equipped with thermal sensors and hardware circuitry that will stop the processor if the temperature increases beyond safe limits.

2.3 Techniques for Reducing Dynamic Power

The dynamic power of a circuit in which all the transistors switch exactly once per clock cycle will be $\frac{1}{2}CV^2F$, if C is the switched capacitance, V is the supply voltage, and F is the clock frequency. However, most of the transistors in a circuit rarely switch from most input changes. Hence, a constant called the activity

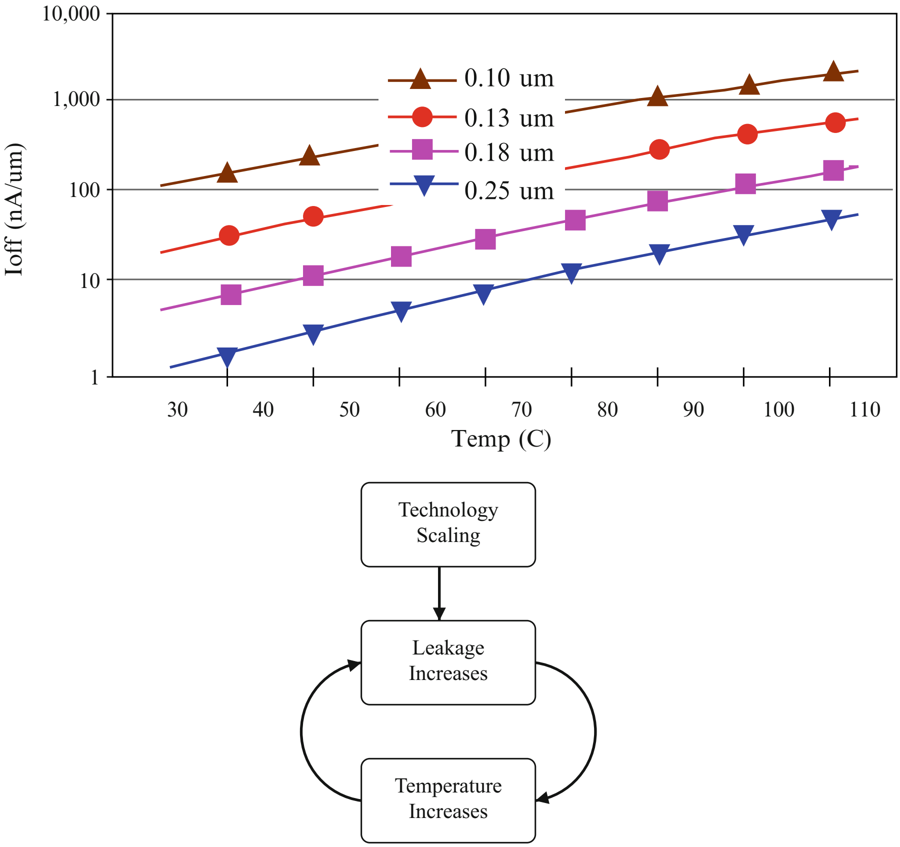


Fig. 2.5 Leakage increases exponentially with temperature. This can create a positive feedback loop, where high power-density increases temperature, which in turn further increases the power-density, causing a thermal runaway

factor ($0 \leq A \leq 1$) is used to model the average switching activity in the circuit. Using A, the dynamic power of a circuit composed of CMOS transistors can be estimated as:

$$P = ACV^2F \tag{2.10}$$

The importance of this equation lies in pointing us towards the fundamental mechanisms of reducing switching power. Figure 2.6 shows that one scheme is by reducing the activity factor A. The question here is: “how to achieve the same functionality by switching only a minimal number of transistors?” Techniques to do this span several design hierarchy levels, right from the synthesis level, where, for example, we can encode states so that the most frequent transitions occur with minimal bit switches, to the algorithmic level, where, for example, changing the sorting algorithm from insertion sort to quick sort, will asymptotically reduce the resulting

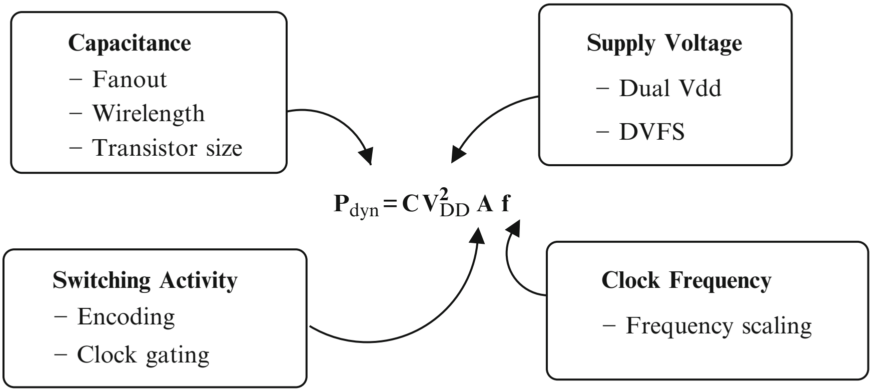


Fig. 2.6 Fundamental techniques to reduce dynamic power

switching activity. The second fundamental scheme is to reduce the load capacitance, C_L. This can be done by using small transistors with low capacitances in non-critical parts of the circuit. Reducing the frequency of operation F will cause a linear reduction in dynamic power, but reducing the supply voltage V_{DD} will cause a quadratic reduction. In the following sections we discuss some of the established and effective mechanisms for dynamic power reduction.

2.3.1 Gate Sizing

The power dissipated by a gate is directly proportional to its capacitive load C_L, whose main components are: i) output capacitance of the gate itself (due to parasitics), ii) the wire capacitance, and iii) input capacitance of the gates in its fanout. The output and input capacitances of gates are proportional to the gate size. Reducing the gate size reduces its capacitance, but increases its delay. Therefore, in order to preserve the timing behavior of the circuit, not all gates can be made smaller; only the ones that do not belong to a critical path can be slowed down.

Any gate re-sizing method to reduce the power dissipated by a circuit will heavily depend on the accuracy of the timing analysis tool in calculating the true delay of the circuit paths, and also discovering false paths. Delay calculation is relatively easier. A circuit is modeled as a directed acyclic graph. The vertices and edges of the graph represent the components and the connection between the components in the design respectively. The weight associated with a vertex (an edge) is the delay of the corresponding component (connection). The delay of a path is represented by the sum of the weights of all vertices and edges in the path. The arrival time at the output of a gate is computed by the length of the longest path from the primary inputs to this gate. For a given delay constraint on the primary outputs, the required time is the time at which the output of the gate is required to be stable. The time slack

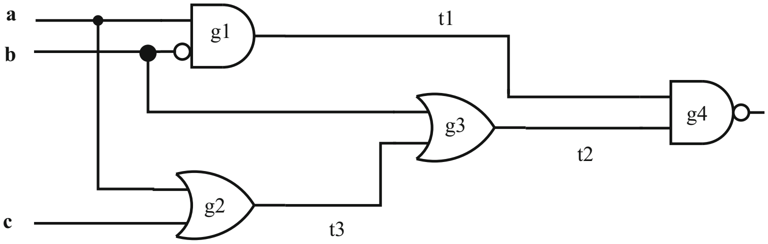


Fig. 2.7 False paths, which do not affect the timing, must be discovered and excluded during timing analysis before performing gate sizing

is defined as the difference of the required time and the arrival time of a gate. If the time slack is greater than zero, the gate can be down-sized. Consider the example circuit in Fig. 2.7. Assuming the delay of AND and OR gates to be 4 units, delay of NOT gate to be 1 unit, and wire delays to be 0, the top 3 timing-critical paths in the circuit are:

$$\begin{aligned} p_1 &: a \rightarrow g_2 \rightarrow g_3 \rightarrow g_4 \rightarrow d \\ p_2 &: b \rightarrow g_1 \rightarrow g_4 \rightarrow d \\ p_3 &: c \rightarrow g_2 \rightarrow g_3 \rightarrow g_4 \rightarrow d \end{aligned}$$

The delay of the path p_1 is 10 units, p_2 is 13 units, and p_3 is 13 units. Therefore static timing analysis will conclude that the gates in both the paths p_2 and p_3 cannot be down-sized. However, by logic analysis, it is easy to figure out that both these seemingly timing critical paths are actually false paths, and can never affect the final stable value of output d. Logically, we have:

$$\begin{aligned} d &= \overline{t1.t2} \\ &= \overline{(a.\overline{b}).(b + t3)} \\ &= \overline{(a.\overline{b}).(b + (a + c))} \\ &= \overline{(a.\overline{b}).(a + b + c)} \\ &= \overline{(a.\overline{b}.a) + (a.\overline{b}.b) + (a.\overline{b}.c)} \\ &= \overline{(a.\overline{b}) + (a.\overline{b}.c)} \\ &= \overline{(a.\overline{b}).(1 + c)} \\ &= \overline{(a.\overline{b})} \end{aligned}$$

Thus, the actual critical path of output d is p_1, which has a delay of only 10 units. Therefore, from the perspective of output d, gates that constitute paths p_2 and p_3, e.g., g_2 and g_3 can be down-sized. The question now is: how to find out if a path is false? A false path is a path that is not *sensitizable*. A path $p = (i, g_0, g_1, ...g_n, o)$ from i to o is sensitizable if a $0 \rightarrow 1$ or $1 \rightarrow 0$ transition at input i can propagate along the entire path p to output o. Whether a change in the inputs will affect the final output at the end of a path however, depends on the values of the other inputs. Since for a given circuit with n primary inputs, there are 2^n possible input vectors, finding out whether a path is sensitizable by enumerating all the input values is clearly infeasible.

To work around this, typically a path-based approach is taken [6, 10, 26]. In order to allow a signal to go through path p from primary input i to the primary output o, we need to set all the other signals feeding to gates along p to be non-control values. The non-control value for an AND (OR) gate is 1 (0). For the path $p_1 = (b, g_1, g_4, d)$, we must set t_2 to be 1, and a to be 1. t_2 can be set to 1 by setting $t3$ to 1, and b to 0. Finally, $t3$ can be set to 1 by setting a to 1, and c can be anything. In this case, there is an assignment of primary inputs that will allow this. On the other hand, if there is no consistent assignment of primary inputs resulting in non-control values of the other inputs of the gates along a path, then the path is non-sensitizable, or a false path. For example, sensitizing path p_3 requires t_1 to be 1, b to be 0, and a to be 0. But this is not possible, because t_1 cannot be 1 with a set to 0. Therefore, path p_3 is a false path.

Taking path sensitizability into account, the calculation of slack time can now be formulated. For an input vector v, let $AT(g_i, v)$ be the arrival time of gate g_i and $RT(g_i, v)$ be the required time of gate g_i under a given delay constraint. The time slack of gate g_i with respect to the input vector v is given by:

$$slack(g_i, v) = RT(g_i, v) - AT(g_j, v)$$

For all input vectors, the slack of gate g_i is defined as:

$$slack(g_i) = min_{\forall v} slack(g_i, v)$$

All gates with slack time greater than zero are candidates for down-sizing. However, to choose a specific one, we must consider several factors, such as the power reduction achievable, and the slack consumption by resizing it. To achieve maximum power reduction, the gate with least delay increment should have higher priority, but to save the total consumed time slack of multiple paths by down-sizing a gate, the gate belonging to a smaller number of paths is preferable for resizing. A combined metric could be:

$$gain(g_i) = \frac{\delta power(g_i)}{\delta delay(g_i) \times |P_{g_i}|}$$

where where $\delta power(g_i)$ is the reduction in power, $\delta delay(g_i)$ is the increase in delay by down-sizing gate g_i, and $|P_{g_i}|$ is the number of noncritical paths passing through gate g_i. The gate with the maximum gain value is selected for resizing [10].

After a gate on a non-critical path is down-sized, a critical path may become non-critical since the load capacitances of some gates on the critical path may decrease. Consequently, the gates on the original critical path may be down-sized. Thus, simultaneous down-sizing and up-sizing multiple gates may reduce power consumption. The satisfaction of delay constraint can be retained if up-sizing gates can compensate the loss in delay caused by downsizing gates on the critical path.

2.3.2 Control Synthesis

Most control circuits are conceived as Finite State Machines (FSM), formally defined as graphs where the nodes represent states, and directed edges, labeled with inputs and outputs, describe the transition relation between states. The state machine is eventually implemented using a state register and combinational logic, that takes in the current state, the current inputs and computes the outputs and the new state, which is then written into the state register at the end of the cycle. The binary values of the inputs and outputs of the FSM are usually determined by external requirements, while the state encoding is left to the designer. Depending on the complexity of the circuit, a large fraction of the power is consumed due to the switching of the state register; this power is very dependent on the selected state encoding. The objective of low power approaches is therefore to choose a state encoding that minimizes the switching power of the state register.

Given a state encoding, the power consumption can be modeled as:

$$P = \frac{1}{2} V_{dd}^2 f \times C_{sr} \times E_{sr}$$

where f is the clock frequency of the state machine, C_{sr} is the effective capacitance of the state register, and E_{sr} is the expected state register switching activity. If S is the set of all states, we can estimate E_{sr} as:

$$E_{sr} = \sum_{i,j \in S} p_{ij} \times h_{ij}$$

where p_{ij} is the probability of a transition between states i and j, and h_{ij} is the Hamming Distance between the codes of states i and j. The best way to estimate p_{ij} is to apply a sufficiently long series of input patterns until the state occurrence and transition probabilities converge towards discrete values [13]. Otherwise, equal probabilities of all input patterns can be assumed, and p_{ij} can be determined stochastically by solving Chapman Kolmogorov equations [4].

Once we have the state transition probabilities, the state encoding problem can be formulated as an embedding of the state transition graph into a Boolean hypercube Fig. 2.8. A Boolean hypercube of dimension n is a graph with 2^n nodes, where every node is labeled with a unique binary value from 0 to $2^n - 1$. Every node v

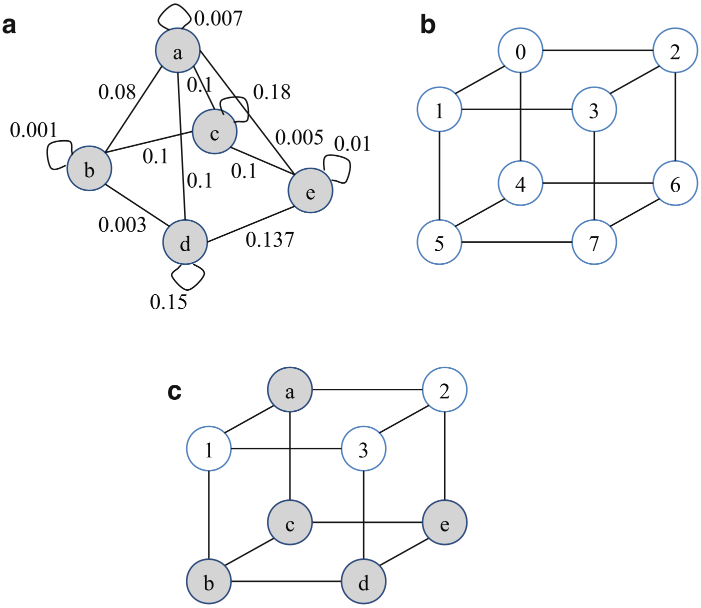


Fig. 2.8 (**a**) State transition graph (STG) contains nodes as states, and switching probabilities as edge weights. The problem of state encoding for low-power is of embedding the STG onto a k-dimensional hypercube. (**b**) A n-dimensional hypercube has 2^n nodes and edges connect nodes that have hamming distance of 1. (**c**) The problem is then to find an injective mapping of nodes from STG to nodes of the hybercube so that the sum of product of distances between nodes and switching frequency is minimized

is connected to n edges labeled $1, ...n$ leading to all nodes whose encodings have a Hamming Distance of 1 from v. Consequently, the hamming distance between any two nodes in the hypercube is equal to the length of the shortest path between the nodes. An embedding of a graph G into a host graph H is an injective mapping of the nodes of G to the nodes of H, so that every edge in G corresponds to the shortest path between the mappings of its terminal nodes in H. The *dilation* of an edge of G is defined as the length of the corresponding path in H.

The dimensionality of a hypercube refers to the number of bits in the state register. To optimize the switching power, a graph embedding with small dilation in a small hypercube must be found. We can always find a low dilation embedding in a hypercube of high dimensionality, but this increases the area and also the power cost of the implementation.

Given the dimensionality of the hypercube, the best solution would be an embedding with dilation 1 for all edges. While such an embedding is possible for cubical graphs, for many graphs it is not possible. The problem of finding an embedding with minimal dilation is NP-complete [7]. The problem of finding the minimum dimension in which a given graph can be embedded, is also NP-complete.

One of the effective solutions for this problem was suggested by Noth et al. [15], in which they create a cubical subgraph of G, which contains the edges with the highest weights, and find a dilation-1 embedding of this subgraph. Since all trees are cubical, a good choice for a subgraph is the maximum spanning tree (MST), which can be easily extracted using greedy algorithms, e.g., Prim's algorithm [9]. Since the dimension of the embedding is strongly connected to the degree of nodes in the tree, tighter embeddings can be found by limiting the degree of any node in the resulting spanning tree. Once the MST is obtained, they embed the MST onto the hypercube using a divide and conquer approach. For this, they first find the center of the tree V_c, E_c with respect to the longest paths. If $p = (v_0, ...v_k)$ is the longest path of the MST, then $V_C^p = \{v_{\lfloor k/2 \rfloor}, v_{\lceil k/2 \rceil}\}$ is the set of nodes in the center of p, where k is the length of longest path p. The center of the tree can then be defined as the set of center nodes of each path. Thus, $V_c = \cup V_c^p$. After picking V_c, we need E_c. It can be shown that every tree has a unique center, essentially proving that there will be either one two nodes in the center of the tree. If there is one node, we can pick an edge of V_c along any of the longest paths; if there are two nodes in V_c, then the two nodes must be connected by an edge, and that edge is our E_c. Removing the edge E_c breaks up the longest path at or near its center, leaving two subtrees of unknown size and structure. Both subtrees are then embedded recursively. Clearly it would be best to balance the subtree embeddings with respect to dimension in order to minimize the dimension of the overall embedding. One approach is to select an edge $(v, w) \in E_c$, whose removal from E leads to the most evenly sized subtrees with respect to the number of edges of the subtrees.

The low power state encoding problem is similar to the classical state encoding problem that targets low area and high performance, but sufficiently different that the resulting encodings will have different properties. Relatively low state register power consumption can be expected.

2.3.3 Clock Gating

Clock signals are omnipresent in synchronous circuits. The clock signal is used in a majority of the circuit blocks, and since it switches every cycle, it has an activity factor of 1. Consequently, the clock network ends up consuming a huge fraction of the on-chip dynamic power. Clock gating has been heavily used in reducing the power consumption of the clock network by limiting its activity factor. Fundamentally, clock gating reduces the dynamic power dissipation by disconnecting the clock from an unused circuit block.

Traditionally, the system clock is connected to the clock pin on every flip-flop in the design. This results in three major components of power consumption:

1. power consumed by combinatorial logic whose values are changing on each clock edge;
2. power consumed by flip-flops – this has a non-zero value even if the inputs to the flip-flops are steady, and the internal state of the flip-flops is constant; and

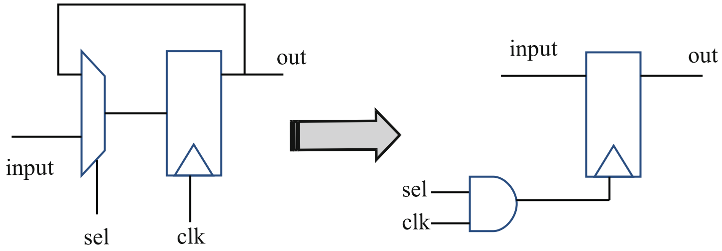


Fig. 2.9 In its simplest form, clock gating can be implemented by finding out the signal that determines whether the latch will have a new data at the end of the cycle. If not, the clock is disabled using the signal

3. power consumed by the clock buffer tree in the design. Clock gating has the potential of reducing both the power consumed by flip-flops and the power consumed by the clock distribution network.

Clock gating works by identifying groups of flip-flops sharing a common *enable* signal (which indicates that a new value should be clocked into the flip-flops). This enable signal is ANDed with the clock to generate the *gated clock*, which is fed to the clock ports of all of the flip-flops that had the common enable signal. In Fig. 2.9, the sel signal encodes whether the latch retains its earlier value, or takes a new input. This sel signal is ANDed with the clk signal to generate the gated clock for the latch. This transformation preserves the functional correctness of the circuit, and therefore does not increase the burden of verification. This simple transformation can reduce the dynamic power of a synchronous circuit by 5–10%.

There are several considerations in implementing clock gating. First, the enable signal should remain stable when clock is high and can only switch when clock is in low phase. Second, in order to guarantee correct functioning of the logic implementation after the gated-clock, it should be turned on in time and glitches on the gated clock should be avoided. Third, the AND gate may result in additional clock skew. For high-performance design with short-clock cycle time, the clock skew could be significant and needs to be taken into careful consideration.

An important consideration in the implementation of clock gating for ASIC designers is the granularity of clock gating. Clock gating in its simplest form is shown in Fig. 2.9. At this level, it is relatively easy to identify the enable logic. In a pipelined design, the effect of clock gating can be multiplied. If the inputs to one pipeline stage remain the same, then all the later pipeline stages can also be frozen. Figure 2.10 shows the same clock gating logic being used for gating multiple pipeline stages. This is a multi-cycle optimization with multiple implementation tradeoffs, and can can save significant power, typically reducing switching activity by 15–25%.

Apart from pipeline latches, clock gating is also used for reducing power consumption in *dynamic* logic. Dynamic CMOS logic is sometimes preferred over

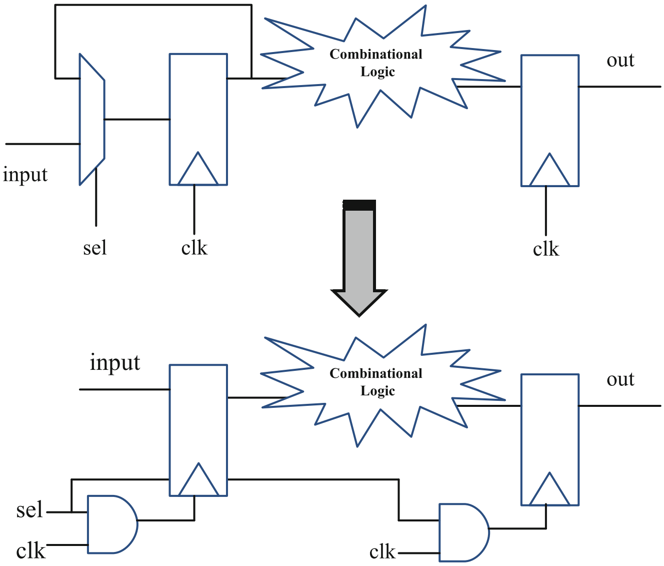


Fig. 2.10 In pipelined designs, the effectiveness of clock gating can be multiplied. If the inputs to a pipeline stage remain the same, then the clock to the later stages can also be frozen

static CMOS for building high speed circuitry such as execution units and address decoders. Unlike static logic, dynamic logic uses a clock to implement the combinational circuits. Dynamic logic works in two phases, *precharge* and *evaluate*. During precharge (when the clock signal is low) the load capacitance is charged. During evaluate phase (clock is high) depending on the inputs to the pull-down logic, the capacitance is discharged.

Figure 2.11 shows the gating technique applied to a dynamic logic block. In Fig. 2.11(a), when the clock signal is applied, the dynamic logic undergoes precharge and evaluate phases (charging the capacitances C_G and C_L) to evaluate the input *In*, so even if the input does not change, the power is dissipated to re-evaluate the same. To avoid such redundant computation, the clock port is gated as shown in Fig. 2.11(b). In this case, when the input does not change or when the output is not used, the gating signal is enabled, which prevents the logic from evaluating the inputs and thereby reduces dynamic power dissipation. An additional *AND* gate is introduced to facilitate clock gating. This additional logic presents its own capacitance and hence dissipates power, but compared to the power saved by preventing the charging of capacitances C_G and C_L (usually large for complex execution units), the *AND* gate power is negligible.

Clock gating at coarse granularity or system level is much more difficult to automate, and designers have to implement it in the functionality themselves. For example, sleep modes in a cell phone may strategically disable the display,

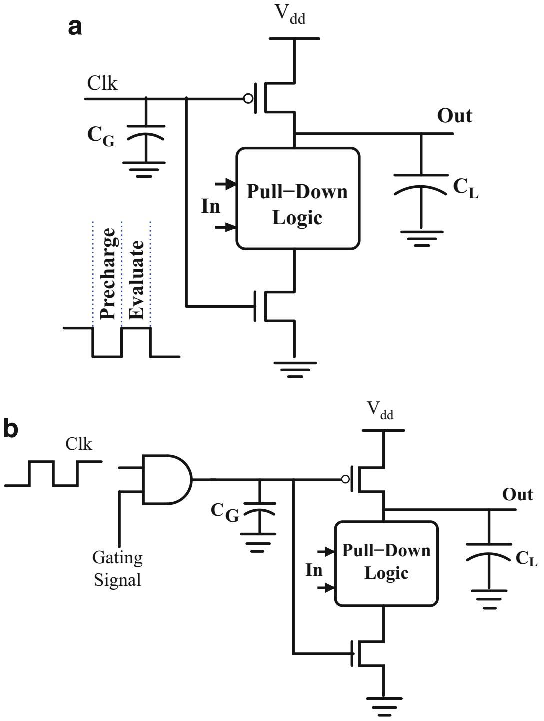


Fig. 2.11 (**a**) Dynamic CMOS Logic (**b**) Clock-gated Dynamic CMOS Logic

keyboard, or radio depending on the phone's current operational mode. System-level clock-gating shuts off entire RTL blocks. Because large sections of logic are not switching for many cycles it has the most potential to save power. However, it may result in inductive power issues due to higher di/dt, since large groups of circuits are turned on/off simultaneously. In contrast, local clock gating is more effective in reducing the worst-case switching power, and also suffers less from di/dt issues. However, local clock gating may lead to frequent toggling of the clock-gated circuit between enable and disable states, as well as higher area, power, and routing overhead, especially when the clock-gating control circuitry is comparable with the clock-gated logic itself.

2.3.4 Voltage and Frequency Scaling

Dynamic power is proportional to the square of the operating voltage. Therefore, reducing the voltage significantly improves the power consumption. Furthermore, since frequency is directly proportional to supply voltage, the frequency of the circuit can also be lowered, and thereby a cubic power reduction is possible. However, the delay of a circuit also depends on the supply voltage as follows.

$$\tau = k.C_L.\frac{V_{dd}}{(V_{dd} - V_t)^2} \tag{2.11}$$

where τ is the circuit delay, k is the gain factor, C_L is the load capacitance, V_{dd} is the supply voltage, and V_t is the threshold voltage. Thus, by reducing the voltage, although we can achieve cubic power reduction, the execution time increases. The main challenge in achieving power reduction through voltage and frequency scaling is therefore to obtain power reduction while meeting all the timing constraints.

Simple analysis shows that if there is slack in execution time, executing as slow as possible, while just meeting the timing constraints is more dynamic-power-efficient than executing as fast as possible and then idling for the remaining time. This is the main idea that is used in exploiting the power reduction that arises due to the cubic relationship with power, and inverse relationship with delay, of the supply voltage.

One approach to recover the lost performance is by scaling down the threshold voltage to the same extent as the supply voltage. This allows the circuit to deliver the same performance at a lower V_{dd}. However, smaller threshold voltages lead to smaller noise margins and increased leakage current. Furthermore, this cubic relationship holds only for a limited range of V_t scaling. The quadratic relationship between energy and V_{dd} deviates as V_{dd} is scaled down into the sub-threshold voltage level. In the sub-threshold region, while the dynamic power still reduces quadratically with voltage, the sub-threshold leakage current increases exponentially with the supply voltage. Hence dynamic and leakage power become comparable in the sub-threshold voltage region, and therefore, "just in time completion" is not energy inefficient. In practice, extending the voltage range below half V_{dd} is effective, but extending this range to sub-threshold operations may not be beneficial.

2.3.4.1 Design-Time Voltage and Frequency Setting

One of the most common ways to reduce power consumption by voltage scaling is that during design time, circuits are designed to exceed the performance requirements. Then, the supply voltage is reduced so as to just meet the performance constraints of the system. This is also called design-time voltage and frequency scaling. Design-time schemes scale and set the voltage and frequency, which remains constant (and therefore inefficient) for all applications at all times.

2.3.4.2 Static Voltage and Frequency Scaling

Systems can be designed for several (typically a few) voltage and frequency levels, and these levels can be switched at run time. In static voltage and frequency scaling, the change to a different voltage and frequency is pre-determined; this is quite popular in embedded systems. However, there are significant design challenges in supporting multiple voltages in CMOS design.

Timing analysis for multiple voltage design is complicated as the analysis has to be carried out for different voltages. This methodology requires libraries characterized for the different voltages used. Constraints are specified for each supply voltage level or operating point. There can be different operating modes for different voltages. Constraints need not be same for all modes and voltages. The performance target for each mode can vary. Timing analysis should be carried out for all these situations simultaneously. Different constraints at different modes and voltages have to be satisfied.

While local on-chip voltage regulation is good way to provide multiple voltages, unfortunately most of the digital CMOS technologies are not suitable for the implementation of either switched mode of operation or linear voltage regulation. A separate power rail structure is required for each power domain. These additional power rails introduce different levels of IR drop, imposing limits on the achievable power efficiency.

2.3.4.3 Dynamic Voltage and Frequency Scaling

The application and system characteristics can be dynamically analyzed to determine the voltage and frequency settings during execution. For example, adaptive scaling techniques make the decision on the next setting based on the recent history of execution.

Voltage and frequency scaling can also be applied to parts of a circuit. Thus, higher voltage can be applied to the timing critical path and modules in other paths may run on lower voltage. This maintains the overall system performance, while significantly reducing power consumption.

Implementing multiple on-chip voltage islands is also a challenge. Signals crossing from one voltage domain to another voltage domain have to be interfaced through *level shifter* buffers which appropriately shift the signal levels. Design of suitable level shifters is a challenging job. The speed at which different power domains switch on or off is also important – a low voltage power domain may activate early compared to the high voltage domain.

Every power domain requires independent local power supply and grid structure and some designs may even have a separate power pad. A separate power pad is possible in flip-chip designs where the power pad can be taken out near from the power domain. For other packaging technologies, the power pads have to be taken out from the periphery, which may impose a limit on the number of power domains.

2.4 Techniques for Reducing Short Circuit Power

Short circuit power is directly proportional to rise time and fall time on gates. Therefore, reducing the input transition times will decrease the short circuit current component. However, propagation delay requirements have to be considered while doing so. Short circuit currents are significant when the rise/fall time at the input of a gate is much larger than the output rise/fall time. This is because the short circuit path will be active for a longer period of time. To minimize the total average short circuit current, it is desirable to have equal input and output edge times. In this case, the power consumed by the short circuit current is typically less than 10% of the total dynamic power. An important point to note is that if the supply is lowered to below the sum of the thresholds of the transistors, $V_{dd} < VT_n + |VT_p|$, the short-circuit currents can be eliminated because both devices will never be on at the same time for any input voltage value.

2.5 Techniques for Reducing Leakage Power

In order to contain the increase in the dynamic power, the supply V_{dd} has undergone a continous reduction in successive technology generations. Along with V_{dd}, V_t must also be scaled down, which results in an exponential increase in leakage power. Consequently, leakage power has become a significant contributor in the total chip power dissipation. Leakage power reduction techniques are especially important for handheld devices such as cell phones, which are "on", but not active most of the time. Consequently, even though such devices dissipate minimal dynamic energy, leakage power becomes a significant contributor in their power equation.

Some of the fundamental techniques to reduce leakage power are discussed in the following sections.

2.5.1 Multiple Supply Voltage

The multiple supply system provides a high-voltage supply for high-performance circuits and a low-voltage supply for low-performance circuits. In a dual V_{dd} circuit, the reduced voltage (low-V_{dd}) is applied to the circuit on non-critical paths, while the original voltage (high-V_{dd}) is applied to the circuit on critical paths. Since the critical path of the circuit is unchanged, this transformation preserves the circuit performance. If a gate supplied with low-V_{dd} drives a gate supplied with high-V_{dd}, the pMOS may never turn off. Therefore a level converter is required whenever a module at the lower supply drives a gate at the higher supply (step-up). Level converters are not needed for a step-down change in voltage. The overhead of level converters can be mitigated by doing conversions at register boundaries and embedding the level conversion inside the latch. Figure 2.12(a) shows a pipeline stage in which some of the paths have low-V_{dd} gates. These are shown in a darker shade in the figure. Notice that some high-V_{dd} gates drive low-V_{dd}, but not vice versa. The transition from low to high V_{dd} is condensed into the level converter latchesshown in the figure. A simple design of level converter latches is shown in Fig. 2.12(b).

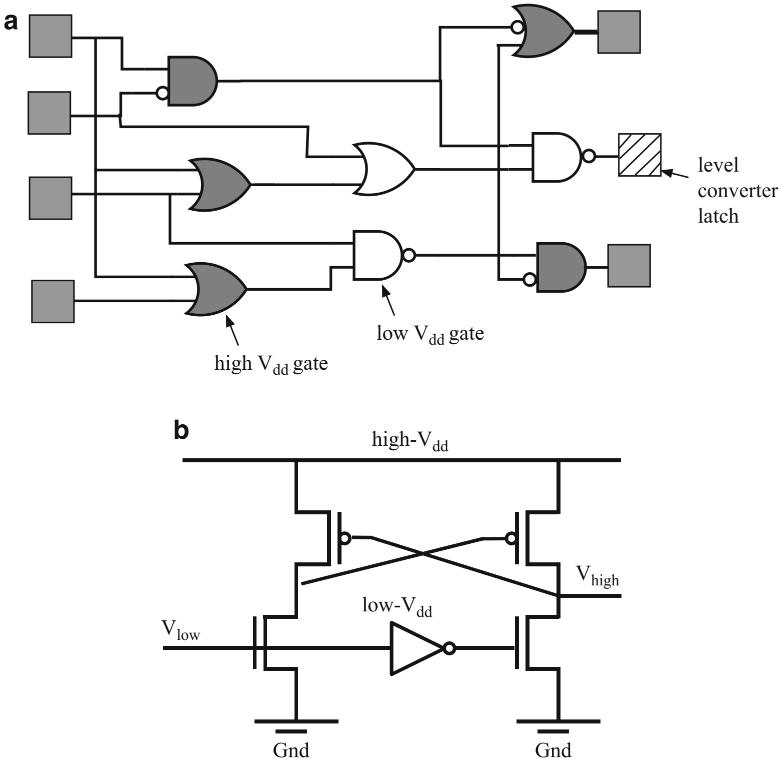


Fig. 2.12 Using multiple V_{dd}s essentially reduces the power consumption by exploiting the slack in the circuit. However, it requires a level converter. (**a**) Multiple supply-voltage pipeline stage. (**b**) Level converter latch

Essentially, the multiple V_{dd} approach reduces power by utilizing excessive slack in a circuit. Clearly, there is an optimum voltage difference between the two V_{dd}s. If the difference is small, the effect of power reduction is small, while if the difference is large, there are few logic circuits that can use low-V_{dd}. Compared to circuits that operate at only high V_{dd}, the power is reduced. The latch circuit includes a level-transition (DC-DC converter) if there is a path where a signal propagates from low V_{dd} logic to high V_{dd} logic.

To apply this technique, the circuit is typically designed using high-V_{dd} gates at first. If the propagation delay of a circuit path is less than the required clock period, the gates in the path are given low-V_{dd}. In an experimental setting [8], the dual V_{dd} system was applied on a media processor chip providing MPEG2 decoding and real-time MPEG1 encoding. By setting high-V_{dd} at 3.3 V and low-V_{dd} at 1.9 V, system power reduction of 47% in one of the modules and 69% in the clock distribution was obtained.

2.5.2 Multiple Threshold Voltage

Multiple V_t MOS devices are used to reduce power while maintaining speed. High speed circuit paths are designed using low-Vt devices, while the high-V_t devices are applied to gates in other paths in order to reduce subthreshold leakage current. Unlike the multiple-V_{dd} transformation, no level converter is required here as shown in Fig. 2.13. In addition, multi-V_t optimization does not change the placement of the cells. The footprint and area of low-V_t and high-V_t cells are similar. This enables timing-critical paths to be swapped by low-V_t cells easily. However, some additional fabrication steps are needed to support multiple V_t cells, which eventually lengthens the design time, increases fabrication complexity, and may reduce yield [1]. Furthermore, improper optimization of the design may utilize more low-V_t cells and hence could end up with increased power!

Several design approaches have been proposed for dual-V_t circuit design. One approach builds the entire device using low-V_t transistors at first. If the delay of a circuit path is less than the required clock period, the transistors in the path are replaced by high-V_t transistors. The second approach allows all the gates to be built with high-V_t transistors initially. If a circuit path cannot operate at a required clock speed, gates in the path are replaced by low-V_t versions. Finally, a third set of approaches target the replacement of groups of cells by high-V_t or low-V_t versions at one go.

In one interesting incremental scheme [18], the design is initially optimized using the higher threshold voltage library only. Then, the multi-V_t optimization computes the power-performance tradeoff curve up to the maximum allowable leakage power limit for the next lower threshold voltage library. Subsequently, the optimization starts from the most critical slack end of this power-performance curve and switches the most critical gate to next equivalent low-V_t version. This may increase the leakage in the design beyond the maximum permissible leakage power. To compensate for this, the algorithm picks the least critical gate from the other end of the power-performance curve and substitutes it with its high-V_t version. If this does not bring the leakage power below the allowed limit, it traverses further from the curve (from

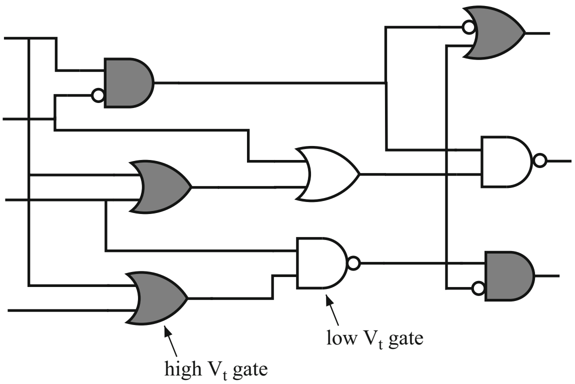


Fig. 2.13 Multiple V_t technology is very effective in power reduction without the overhead of level converters. The white gates are implemented using low-V_t transistors

least critical towards most critical) substituting gates with high-V_t gates, until the leakage limit is satisfied. Then the algorithm continues with the second most critical cell and switches it to the low-V_t version. The iterations continue until we can no longer replace any gate with the low-V_t version without violating the leakage power limit. The multi-V_t approach is very effective. In a 16-bit ripple-carry adder, the active-leakage current was reduced to one-third that of the all low-V_t adder [1].

2.5.3 Adaptive Body Biasing

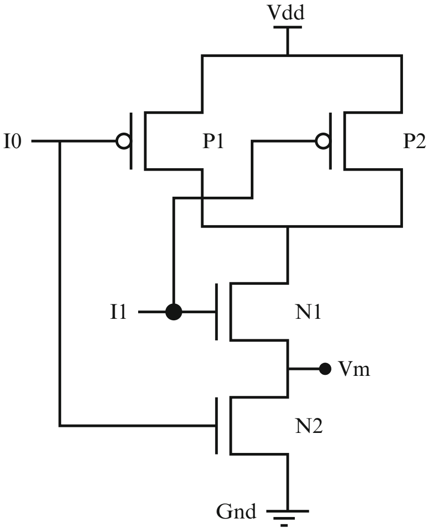
One efficient method for reducing power consumption is to use low supply voltage and low threshold voltage without losing performance. But increase in the lower threshold voltage devices leads to increased sub threshold leakage and hence more standby power consumption. One solution to this problem is *adaptive body biasing (ABB)*. The substrate bias to the n-type well of a pMOS transistor is termed V_{bp} and the bias to the p-type well of an nMOS transistor is termed V_{bn}. The voltage between V_{dd} and V_{bp}, or between *GND* and V_{bn} is termed V_{bb}. In the active mode, the transistors are made to operate at low-V_{dd} and low-V_t for high performance. The fluctuations in V_t are reduced by an adaptive system that constantly monitors the leakage current, and modulates V_{bb} to force the leakage current to be constant. In the idle state, leakage current is blocked by raising the effective threshold voltage V_t by applying substrate bias V_{bb}.

The ABB technique is very effective in reducing power consumption in the idle state, with the flexibility of even increasing the performance in the active state. While the area and power overhead of the sensing and control circuitry are shown to be negligible, there are some manufacturing-related drawbacks of these devices [20]. ABB requires either twin well or triple well technology to achieve different substrate bias voltage levels in different parts of the IC. Experiments applying ABB to a discrete cosine transform processor reported a small 5% area overhead. The substrate-bias current of V_{bb} control is less than 0.1% of the total current, a small power penalty.

2.5.4 Transistor Stacking

The subthreshold leakage current flowing through a stack of transistors connected in series, is reduced if atleast one of them is switched *off*. For example, consider the NAND gate schematic in Fig. 2.14. When both N1 and N2 are turned off, the voltage V_m at the intermediate node between N1 and N2 is positive due to the small drain current. This has the following three effects on subthreshold leakage current of the circuit:

- Due to the positive potential V_m, the gate to source volatage of transistor N1 $(0 - V_m = -V_m)$ becomes negative, resulting in reduced subthreshold current of N1.



Inputs (I0 I1)	Leakage
00	2.I$_{leak}$
01	I$_{leak}$
10	I$_{leak}$
11	~0

Fig. 2.14 Several gates in a typical CMOS circuit are automatically stacked, and in low-leakage state. The problem of finding low-leakage input vector is to find the inputs that will put most of the transistors in a low-leakage state

- The positive volatage V_m acts as a body bias to increase the threshold voltage of N1, again resulting in reducing the subthreshold leakage.
- The drain to source potential of N1 increases due to positive potential V_m, resulting in further increase in the threshold voltage, thereby decreasing the leakage current.

Due to this *stacking effect*, leakage power of a CMOS gate depends heavily on the input vector. For example, consider the NAND gate schematic in Fig. 2.14. When the inputs *I0 I1* are "00", both P1 and P2 are on, and N1 and N2 are off. Therefore, $I_{leak}^{00} = I_{N1} + I_{N2} \approx 2 * I_{leak}$. When the inputs are "01", N1 is off, but N2 is on. N1 can be treated as shorted, and its leakage current is ignored. Also, since P1 is on and P2 is off, $I_{leak}^{01} = I_{P2} = I_{leak}$. Similarly, when the inputs are "10", $I_{leak}^{10} = I_{P1} = I_{leak}$. Finally, when inputs are "11", N1 and N2 are on, but P1 and P2 are off. Due to the stacking effect, $I_{leak}^{11} \ll I_{leak}$.

Typically, several gates in a large circuit block may already have low leakage due to the stacking effect. Input vector selection seeks to further reduce the leakage of an idle circuit by applying an input vector that gives as small a leakage current as possible.

The problem of finding the low leakage input vector is motivated by demonstrating that the leakage power of circuits can vary by an order of magnitude over a range of randomly chosen input vectors. The idea is that by using minimal additional circuitry, the logic can be modified so that whenever a circuit is in standby mode, its internal state is set to low leakage. When the circuit is reactivated, it is returned to its last valid state. If the idle periods are long enough, this leads to significant power reductions.

2.5.5 Power Gating

Power Gating is an extremely effective scheme for reducing the leakage power of idle circuit blocks. The power (V_{dd}) to circuit blocks that are not in use is temporarily turned off to reduce the leakage power. When the circuit block is required for operation, power is supplied once again. During the temporary shutdown time, the circuit block is not operational – it is in *low power* or *inactive* mode. Thus, the goal of power gating is to minimize leakage power by temporarily cutting-off power to selective blocks that are not active.

As shown in Fig. 2.15, power gating is implemented by a pMOS transistor as a header switch to shut off power supply to parts of a design in standby or sleep mode. nMOS footer switches can also be used as sleep transistors. Inserting the sleep transistors splits the chip's power network into two parts: a *permanent power*

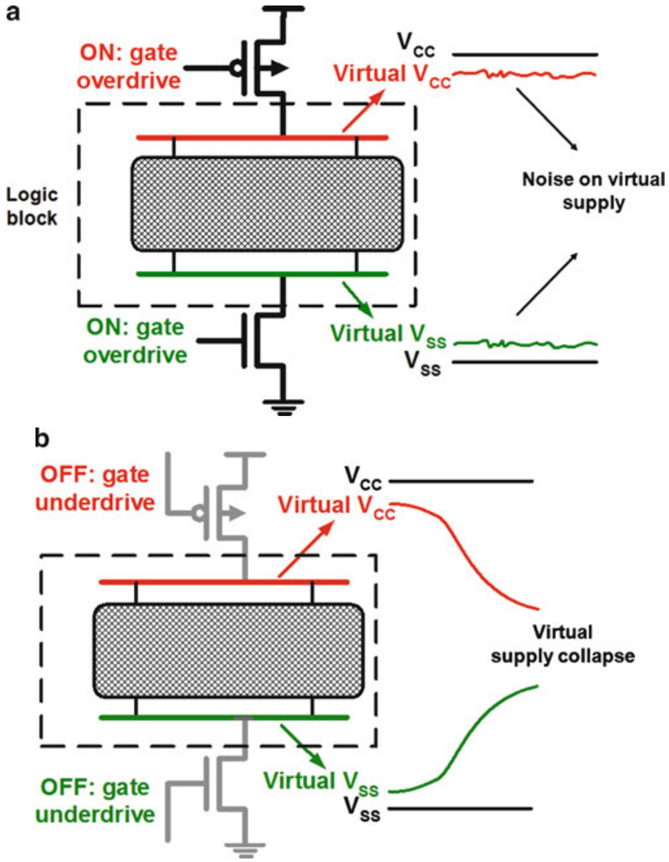


Fig. 2.15 (**a**) Active mode: in the "on" state, the circuit sees a virtual V_{cc} and virtual V_{ss}, which are very close to the actual V_{cc}, and V_{ss} respectively. (**b**) Idle mode: in the "off" state, both the virtual V_{cc} and virtual V_{ss} go to a floating state.

network connected to the power supply and a *virtual power network* that drives the cells and can be turned off.

The biggest challenge in power gating is the size of the power gate transistor. The power gate size must be selected to handle the required amount of switching current at any given time. The gate must be big enough such that there is no measurable voltage (IR) drop due to it. Generally, we use 3X the switching capacitance for the gate size as a rule of thumb.

Since the power gating transistors are rather large, the slew rate is also large, and it takes more time to switch the circuit on and off. This has a direct implication on the effectiveness of power gating. Since it takes a long time for the power-gated circuit to transition in and out of the low power mode, it is not profitable to power gate large circuits for short idle durations. This implies that either we implement power gating at fine granularity, which increases the overhead of gating, or find large idle durations for coarse-grain power gating, which are fewer and more difficult to discover. In addition, coarse-grain power gating results in a large switched capacitance, and the resulting rush current can compromise the power network integrity. The circuit needs to be switched in stages in order to prevent this. Finally, since power gates are made of active transistors, the leakage of the power gating transistor is an important consideration in maximizing power savings.

For fine-grain power-gating, adding a sleep transistor to every cell that is to be turned off imposes a large area penalty. Fine-grain power gating encapsulates the switching transistor as a part of the standard cell logic. Since switching transistors are integrated into the standard cell design, they can be easily be handled by EDA tools for implementation. Fine-grain power gating is an elegant methodology resulting in up to 10X leakage reduction.

In contrast, the coarse-grained approach implements the grid style sleep transistors which drive cells locally through shared virtual power networks. This approach is less sensitive to process variations, introduces less IR-drop variation, and imposes a smaller area overhead than the fine-grain implementations. In coarse-grain power gating, the power-gating transistor is a part of the power distribution network rather than the standard cell.

2.6 Summary

Power considerations led to the replacement of bipolar logic by CMOS in the 1980s in spite of the former resulting in smaller and faster devices. Continuous advacements in CMOS technology enabled an exponential the scaling down of transistor area and scaling up of switching speed over the decades. Unfortunately, the power density increased prominently as a consequence. Leakage currents are expected to result in further increases in power density for technology nodes below 45 nm, which is becoming unsustainable with currently available solutions. This sets up the stage for investigation into aggressive power reduction techniques at every level of design abstraction.

Through this chapter we have introduced various fundamental device and circuit level techniques as well as power management techniques for low power CMOS technology. We discussed various sources of power consumption in a CMOS transistor. We introduced the major components of power: dynamic, short-circuit, and leakage, followed by the basics of device and circuit level techniques to reduce these power components. Techniques for power optimizations at higher levels of design abstraction (microarchitectural, compiler, operating systems, etc.), discussed in later chapters, build upon the foundations laid here.

References

1. Agarwal, A., Kang, K., Bhunia, S.K., Gallagher, J.D., Roy, K.: Effectiveness of low power dual-Vt designs in nano-scale technologies under process parameter variations. In: ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design, pp. 14–19. ACM, New York, NY, USA (2005). DOI http://doi.acm.org/10.1145/1077603.1077609
2. Brews, J.R.: The Submicron MOSFET, Chapter 3 in S.M. Sze, editor, High Speed Semiconductor Devices. John Wiley & Sons, New York (1990)
3. Choi, Y.K., Ha, D., King, T.J., Bokor, J.: Investigation of gate-induced drain leakage (gidl) current in thin body devices: single-gate ultra-thin body, symmetrical double-gate, and asymmetrical double-gate mosfets. Japan Journal of Applied Physics part I pp. 2073–2076 (2003)
4. Cox, D., Miller, H.: The Theory of Stochastic Processes. Chapman Hall (1965)
5. Davidson, E.: Packaging technology for the IBM 3090 series systems. In: IEEE Computer Society Spring Workshop (1985)
6. Du, D.H., Yen, S.H., Ghanta, S.: On the general false path problem in timing analysis. In: DAC '89: Proceedings of the 26th ACM/IEEE Design Automation Conference, pp. 555–560. ACM, New York, NY, USA (1989). DOI http://doi.acm.org/10.1145/74382.74475
7. Garey, M.R., Johnson, D.S.: Computers and Intractibility – A Guide to the Theory of NP-Completeness. W.H. Freeman (1979)
8. Ichiba, F., Suzuki, K., Mita, S., Kuroda, T., Furuyama, T.: Variable supply-voltage scheme with 95 In: ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design, pp. 54–59. ACM, New York, NY, USA (1999). DOI http://doi.acm.org/10.1145/313817.313849
9. Lengauer, T.: Combinatorial Algorithms for Integrated Circuit Layout. Verlag (1990)
10. Lin, H.R., Hwang, T.T.: Power reduction by gate sizing with path-oriented slack calculation. In: Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95., IFIP International Conference on Hardware Description Languages; IFIP International Conference on Very Large Scale Integration., Asian and South Pacific, pp. 7–12 (1995). DOI 10.1109/ASPDAC.1995.486194
11. Lyman, J.: Supercomputers demand innovation in packaging and cooling. Electronics Maganize pp. 136–143 (1982)
12. Moore, G.E.: Cramming more components onto integrated circuits. Electronics Maganize **38**(8) (1965)
13. Najm, F.N., Goel, S., Hajj, I.N.: Power estimation in sequential circuits. In: DAC '95: Proceedings of the 32nd annual ACM/IEEE Design Automation Conference, pp. 635–640. ACM, New York, NY, USA (1995). DOI http://doi.acm.org/10.1145/217474.217602
14. Neamen, D.A.: Semiconductor Physics and Devices: Basic Principles. Tata McGraw Hill Publishing Company (1992)
15. Nöth, W., Kolla, R.: Spanning tree based state encoding for low power dissipation. In: DATE '99: Proceedings of the conference on Design, automation and test in Europe, p. 37. ACM, New York, NY, USA (1999). DOI http://doi.acm.org/10.1145/307418.307482

16. Oktay, S.., Kammerer, H.C.: A conduction-cooled module for high performance LSI devices. IBM Journal of Research and Development **26**(1), 55–56 (1982)
17. Pierret, R.F.: Semiconductor Device Fundamentals. Addison-Wesley, Reading, MA (1996)
18. Puri, R.: Minimizing power under performance constraint. In: International Conference on Integrated Circuit Design and Technology, 2004. ICICDT ’04., pp. 159–163 (2004)
19. Ramadhyani, S.., Incropera, F.P.: Forced convection cooling of discrete heat sources with or without surface enhancement. In: International Symposium on Cooling Technology for Electronic Equipment, pp. 249–264 (1987)
20. Tsividis, Y.P.: Operation and Modeling of the MOS Transistor. McGraw-Hill, New York (1987)
21. Tuckerman, D., Pease, F.: High-performance heat sinking for VLSI. IEEE Electron Device Letters **EDL-2**(5), 126–129 (1981)
22. Veendrick, H.: Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits. Solid-State Circuits, IEEE Journal of **19**(4), 468–473 (1984)
23. Watari, T., Murano, H.: Packaging technology for the NEX SX supercomputer. In: Electronic Components Conference, pp. 192–198 (1985)
24. Wu, P., Little, W.A.: Measurement of the heat transfer characteristics of gas flow in fine channel heat exchangers used for microminiature refrigerators (1984)
25. Yamamoto, H., Udagawa, Y., Okada, T.: Cooling and packaging technology for the FACOM M-780. Fujistsu Journal **37**(2), 124–134 (1986)
26. Yen, H.C., Ghanta, S., Du, H.C.: A path selection algorithm for timing analysis. In: DAC ’88: Proceedings of the 25th ACM/IEEE Design Automation Conference, pp. 720–723. IEEE Computer Society Press, Los Alamitos, CA, USA (1988)

Chapter 3
Power-efficient Processor Architecture

3.1 Introduction

Since the creation of the first processor/CPU in 1971, silicon technology consistently allowed to pack twice the number of transistors on the same die every 18 to 24 months [33]. Scaling of technology allowed the implementation of faster and larger circuits on silicon, permitting a sophisticated and powerful set of features to be integrated into CPU. Figure 3.1 shows the evolution of processors from 4-bit scalar datapath to 64-bit superscalar datapath and their respective transistor counts. Processors evolved not only in terms of datapath width, but also in terms of a wide variety of architectural features such as pipelining, floating point support, on-chip memories, superscalar processing, out-of-order processing, speculative execution, multi-threading, muticore CPUs, etc.

Microprocessors have evolved through the generations with performance (processing capacity) improvement as their primary objective. Technology scaling, along with architectural techniques to enhance parallel processing capabilities of a processor, have boosted the performance from a few instructions per second to a few billion instructions per second, as shown in Fig. 3.2. Primitive processors were scalar, where each instruction could process only one data element at a time. Since the introduction of the first processor in early 1970s to the mid of 1980s, the additional transistors provided by technology scaling were essentially used to improve the *bit-level parallelism* by increasing the word (datapath) width from 4-bit to 32-bit.

From the early 1990s, much of the increase in processing capacity was derived by exploiting *Instruction-Level Parallelism* (ILP) [23]. Pipelined architectures and on-chip cache memories have considerably improved the processor throughput. Figure 3.3 shows a standard five-stage pipelined architecture of a processor datapath. The processing of an instruction is split into five independent stages (Fetch, Decode, Execute, Memory, and Writeback) separated by pipeline registers. A pipelined datapath can be clocked at a much higher frequency by dividing the work among multiple clock cycles. Though the processing of an instruction requires at least five clock cycles, the execution of up to five consecutive instructions can be overlapped as shown in Fig. 3.4(a), allowing the CPU to achieve a maximum throughput of one instruction per cycle.

P.R. Panda et al., *Power-efficient System Design*, DOI 10.1007/978-1-4419-6388-8_3,

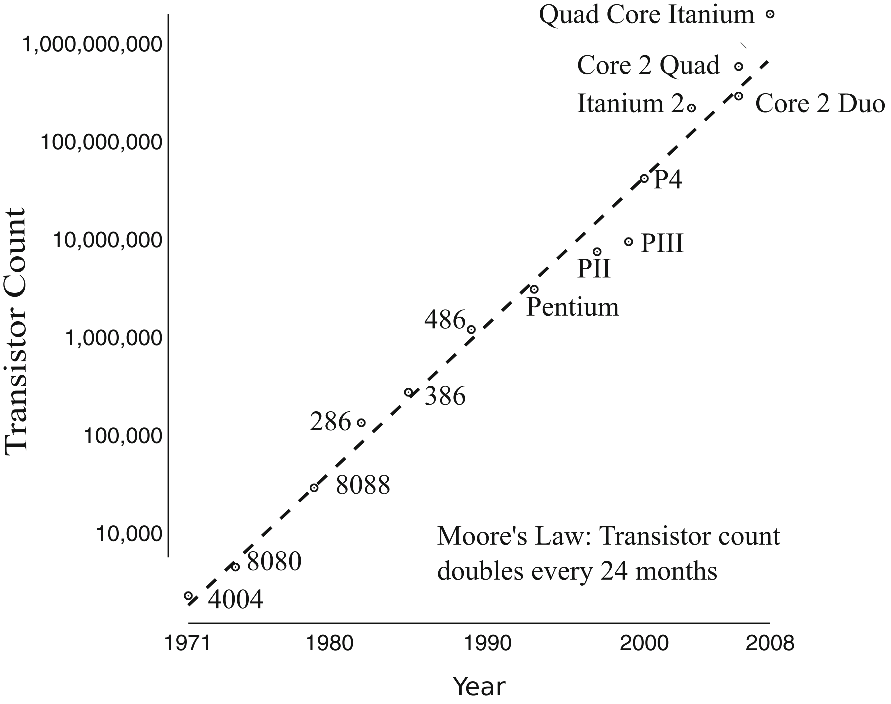


Fig. 3.1 Moore's law and the evolution of processors [1]: With transistor density doubling every 24-months, the additional transistors were used for building more sophisticated processors

ILP was further exploited in the *superscalar* architectures by using multiple function units to allow parallel execution of instructions in each cycle. This is achieved by simultaneously dispatching a set of instructions to the execution units. To support multiple issue, a dispatch logic is added as a pipeline stage after decoding, which analyzes the dependencies among the instructions and issues independent instructions to the available function units. The working of an example two issue pipeline is shown in Fig. 3.4(b). The pipeline is designed to fetch two instructions per cycle and can provide a maximum throughput of two instructions per clock cycle. However, due to data and control dependencies and resource conflicts, the actual throughput achieved by the processor is lower. Also, pipelined implementation incurs the overhead of additional registers in the critical path, which increases instruction execution latency. For an in-order processor, pipeline stalls represent a major bottleneck in achieving the desired throughput. Stalling occurs due to three types of hazards: (i) data, (ii) structural, and (iii) control. Pipelined execution can potentially lead to race conditions when the instructions have data dependencies such as Read-After-Write (RAW), Write-After-Write (WAW) or Write-After-Read (WAR). Such dependencies are identified and the execution of these (and subsequent) instructions is stalled until they are resolved. Structural hazards occur when multiple instructions require the same hardware unit in a given cycle. A control hazard occurs due to branch

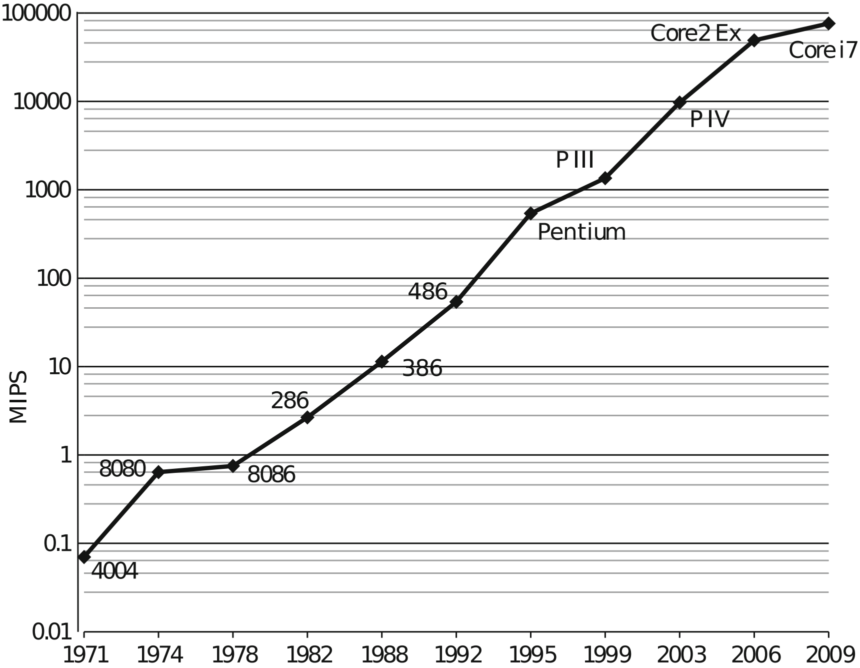


Fig. 3.2 Processing capacity of mainstream Intel processors [1]: technology scaling (leading to faster transistors) along with advanced architectural techniques for extracting bit-level parallelism (4-bit to 64-bit datapath), instruction-level parallelism (pipelining, superscalar, out-of-order processing support), and thread-level parallelism (hyper-threading, multi-core) drives the performance scaling in processors

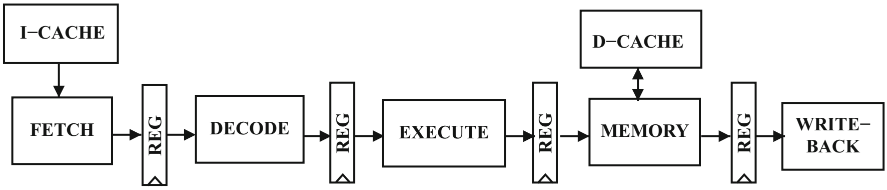


Fig. 3.3 Five-stage pipelined processor datapath: pipeline stages are separated by registers and hence, can act independently on different elements. Partitioning also allows us to clock the circuit at higher frequencies. On-chip memories provide low latency access to instructions and data

instructions, since the instructions following the branch cannot be fetched until the branch is resolved, resulting in a pipeline stall.

Pipeline hazards are alleviated in modern processors through dynamic out-of-order scheduling and by using multiple execution units. In dynamic scheduling, though instructions are fetched and graduated in program order they are allowed to be executed out-of-order. Sophisticated techniques such as register renaming and

a

cycle	1	2	3	4	5	6	7	8	9
Inst−1	F	D	E	M	W				
Inst−2		F	D	E	M	W			
Inst−3			F	D	E	M	W		
Inst−4				F	D	E	M	W	
Inst−5					F	D	E	M	W

b

cycle	1	2	3	4	5	6	7
Inst−1	F	D	E	M	W		
Inst−2	F	D	E	M	W		
Inst−3		F	D	E	M	W	
Inst−4		F	D	E	M	W	
Inst−5			F	D	E	M	W

Fig. 3.4 Working of a five-stage (F-Fetch, D-Decode, E-Execute, M-Memory and W-Writeback) pipelined processor. An N-issue processor can attain a maximum throughput of N instructions per cycle. However, in reality, it is limited by the amount of parallelism available in the instruction stream. (**a**) Single-issue pipeline. (**b**) Dual-issue/superscalar pipeline

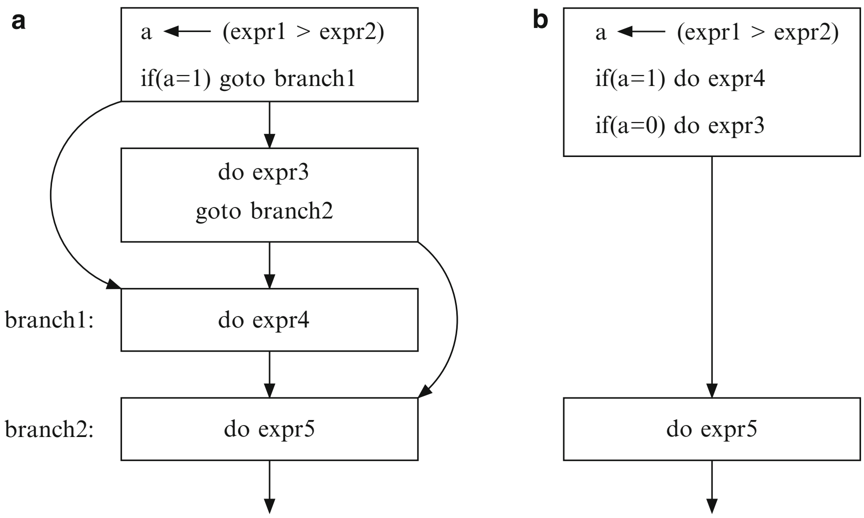


Fig. 3.5 Use of predication: Predicated instructions can be used to replace branches in the instruction flow. (**a**) Without predicate instruction. (**b**) With predicate instruction

speculative execution are used to eliminate data (WAR,WAW) dependencies and most of the control dependencies. Control dependencies are resolved either using branch predictors or through predicated instructions. Advanced branch prediction units predict the branch direction with less than 4% error. Predicated instructions are used to remove branches as shown in the example in Fig. 3.5. An additional bit, called predicate bit, is used for storing the result of conditions. Predicated instructions are fetched, decoded, and may also be executed like any other instruction. However, the results of the instructions whose predication bit is evaluated to be false, are discarded. These techniques help in efficient utilization of the pipeline resources and ultimately result in the processing of more instructions per clock cycle. However, the above techniques substantially increase the hardware complexity and hence the power dissipation in a processor.

3.1.1 Power Budget: A Major Design Constraint

The frequency of various mainstream processors from Intel Corporation is shown in Fig. 3.6(a). Technology scaling and increasing pipeline depths helped improve the processor frequency from 108 KHz in 1971 to almost 4 GHz in 2005. Increasing frequency also saw a rise in power dissipation, shown in Fig. 3.6(b). With cheap cooling solutions available to disperse heat generated by a processor, power dissipation was never considered a prominent design constraint in a yesteryear processors. However, beyond the $0.13\mu m$ process node [9], with further scaling of CMOS technology and increasing processor complexity (transistor count), the power densities have risen exponentially to an extent that they have hit the *Power Wall*. Since the existing cooling solutions are incapable of handling such power densities in a cost-effective manner, power dissipation has become a major constraint in designing modern processors.

Modern processors dissipate more than 100 watts of power, and a stage has been reached where power dissipation is limiting further frequency scaling. In order to maintain the performance scaling of the processor, the additional transistors provided by technology scaling are now used to exploit *thread-level parallelism* leading to an era of chip-multiprocessor (CMP) architectures. Compared to frequency scaling, exploiting thread-level parallelism using CMP architectures has provided better *performance per watt*. This trend is observed from Figs. 3.2 and 3.6, where the frequency of processors has been reduced after 2005 while the number of cores has increased [40]. To further reduce the overall power consumption, each core in a CMP is designed to be power-efficient.

Increasing constraints on power budgets have created the need for low power architectures without compromising much on the performance front. This section briefly describes the functionality of various hardware blocks in a typical out-of-order superscalar datapath to give an idea of their complexity and hence power consumption.

3.1.1.1 Why does Parallel Processing Reduce Power?

Multiple processing elements, in conjunction with parallel algorithms, can be used for reducing the computation time of a task. However, our objective here is to reduce power consumption. For a given computation time, we can easily establish that compared to sequential processing, parallel processing results in a more power-efficient solution. Power consumed by a capacitive system is given by:

$$\text{Power} \propto \text{Voltage}^2 \times \text{clock frequency}$$

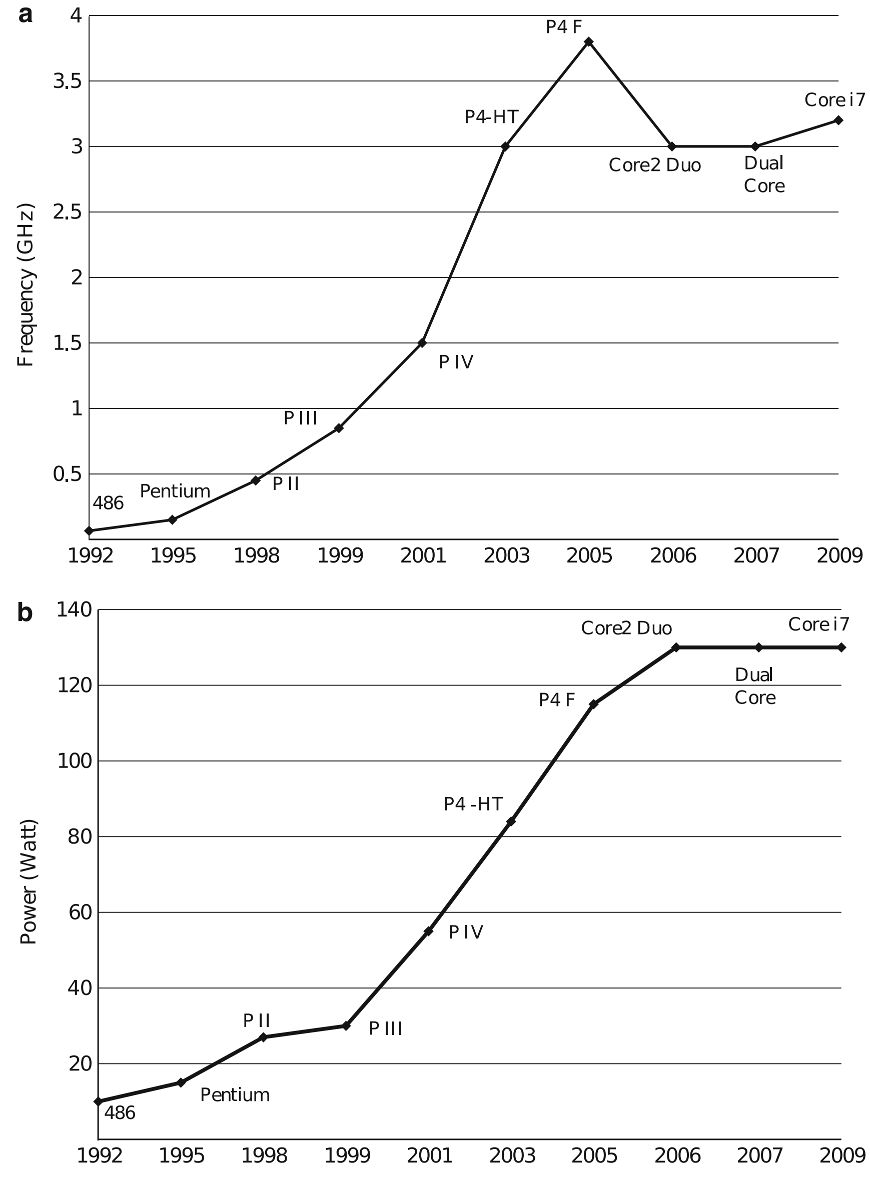


Fig. 3.6 Frequency and power dissipation of mainstream Intel processors [1]: for decades, higher performance in processors always meant higher frequency. With exponential rise in power dissipation and no efficient cooling solutions around, clock frequencies could not scale further. This led to the drop in clock frequencies and evolution of multi-core architectures to keep up the performance scaling. (**a**) Frequency scaling of processors between 1992–2009. (**b**) Power dissipation of processors between 1992–2009

In a parallel system, if N processors are employed to perform the same task in the same amount of time, then the speed at which each processor needs to run could be decreased by a factor N. Hence, the power consumed by this system is given by,

$$\text{Power} \propto N \times \text{Voltage}_{new}^2 \times \text{clock frequency}_{new}$$

The relationship between gate delay and the applied voltage for CMOS logic can be expressed as:

$$\text{Gate Delay} \propto \frac{\text{Voltage}}{(\text{Voltage} - V_{th})^2}$$

To a first order of approximation, circuit speed can be considered proportional to the voltage applied. Hence, if the speed of the circuit needs to be divided by a factor N, then its voltage has to be divided by a factor N. Thus, when a parallel system with N processors is used for performing the same task in the same amount of time, an individual processor's voltage and frequency are scaled down by N times. Since power varies as $V^2 f$, the power consumed by one processor is $\frac{1}{N^2} \times \frac{1}{N} = \frac{1}{N^3}$ times the power of the original processor. As there are N processors in an N-way parallel system, the total system power is $\frac{1}{N^3} \times N = \frac{1}{N^2}$ times the power consumed by the single processor system.

Consider a task A, which needs to be completed within time period T as shown in Fig. 3.7. We have two different systems, System-1 and System-2, with one and two processors respectively. System-1 can finish the task A in time T by working at a frequency F. Assuming the task can be parallelized, System-2 (with two processors)

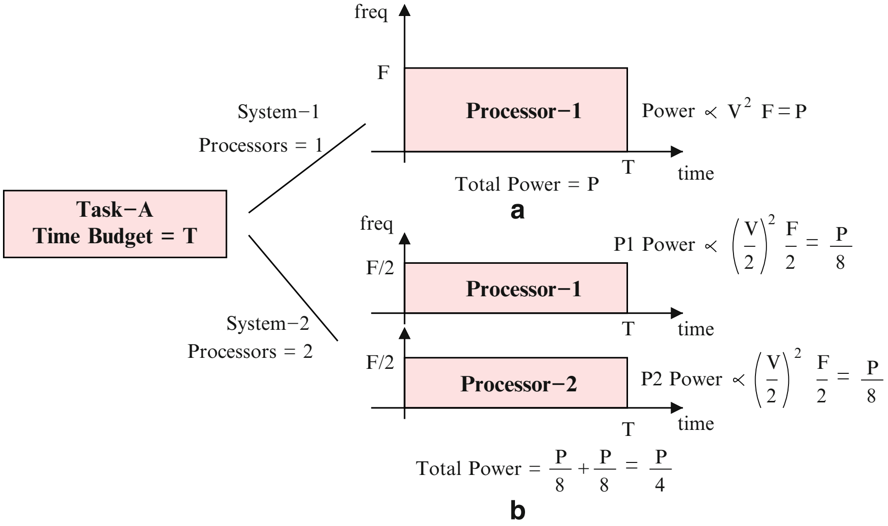


Fig. 3.7 Parallel processing reduces power: (**a**) the system with single processor working at frequency F consumes more power than (**b**) the parallel voltage-scaled system with two processors working at frequency F/2. Both systems have have the same computational capacity

needs to work at only half the frequency of System-1 (i.e. $\frac{F}{2}$) to complete the same task A in time T. Thus, if System-1 consumes P watts in processing task A, the two processors in System-2 would consume $\frac{P}{4} \times \frac{1}{2} = \frac{P}{8}$ watts each, leading to a total power of $\frac{P}{8} \times 2 = \frac{P}{4}$. Thus, System-2 consumes $\frac{1}{4}$ the power of System-1.

In effect, through parallel processing, it is possible to improve overall performance without increasing the clock frequency and, in fact, reduce power at the same time. This single observation has profoundly impacted the evolution of processor architecture, paving the path to CMP/multicore systems. In a CMP, the workload is divided among the cores for power efficient processing.

3.1.2 Processor Datapath Architecture

In an N-way superscalar processor, up to N-instructions can be dispatched in a cycle. Performance improvement in modern processors is achieved mainly through multiple-issue logic, dynamic scheduling, and techniques such as renaming (to resolve false dependencies) and speculative execution through branch prediction. When instructions are processed in an order different from the actual program order, unpredictable situations could arise due to exceptions. For example, in an out-of-order machine, the completion or graduation of all instructions before a faulting instruction is not guaranteed. Consider the following sequence of instructions which are ready to execute:

1. DIV REG0, REG1, REG2
2. ADD REG6, REG5, REG6

As the instructions are independent, both can be issued to execution units simultaneously. Since the latency of the division operation is larger than that of addition, the *ADD* instruction would complete prior to the completion of *DIV* instruction and the register *REG6* would be updated. If an exception occurs during the division operation (say, division-by-zero) after the completion of addition, then the machine state cannot be restored after the exception as one of the operands of the addition instruction would be destroyed already. Hence, it is important for dynamic issue machines to support precise exceptions and preserve exception behavior as in sequential in-order machines. These requirements make it obligatory for the processor architecture to maintain proper state (program counter, register file, memory) during exceptions.

Figure 3.8 shows the block diagram of a superscalar datapath [20]. The dataflow and functionality of hardware blocks are explained in terms of various pipeline stages discussed here.

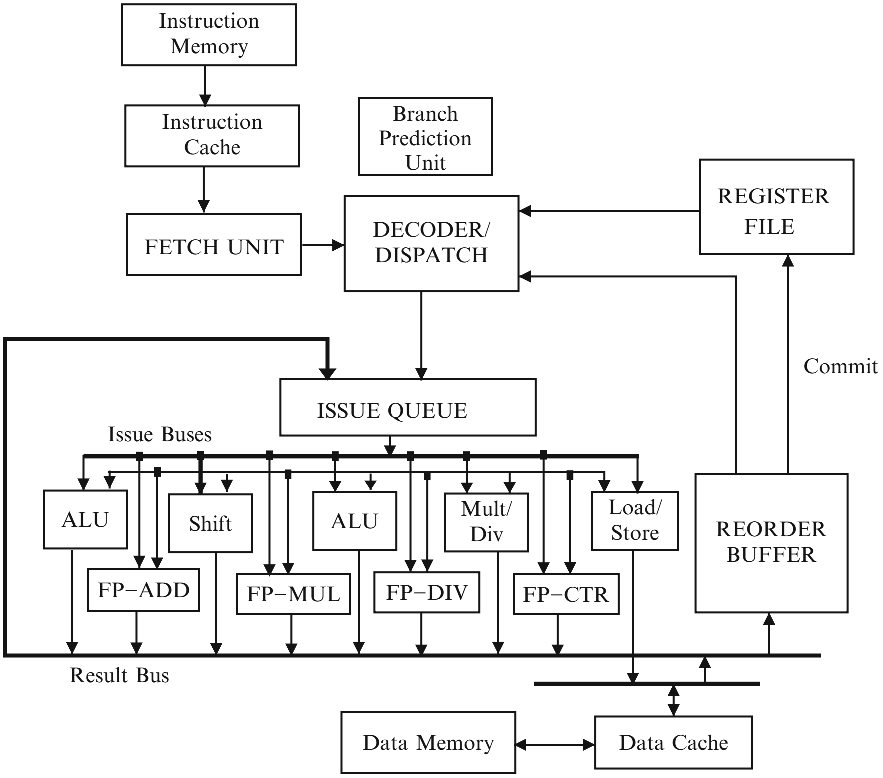


Fig. 3.8 Datapath of a superscalar processor. Front end stages of the pipeline can fetch and decode multiple instructions per cycle. The decoded instructions are then dispatched to the issue queue. With the availability of various function units along with out-of-order execution support for parallel execution, multiple instructions can be issued in parallel. The completed instructions are collected into a re-order buffer and later committed in program order

3.1.2.1 Instruction Fetch

The fetch unit reads the instruction stream from the instruction cache and passes them to the decoder. The address of an instruction is given by a special register called the Program Counter (PC), which is either derived from the address predicted by the branch prediction unit or from the next PC value.

A block diagram of the fetch unit is shown in Fig. 3.9. The PC value is updated either by the PC update unit or the branch prediction unit. In the normal flow of instructions, the address of the next instruction is derived by incrementing the current PC value, which is computed by the PC update unit. In case of branch instruction, the address of the next instruction is specified by the branch target location. The branch prediction unit specifies the next PC value when such instructions are encountered. The value of PC is used to fetch the cache block from the instruction cache.

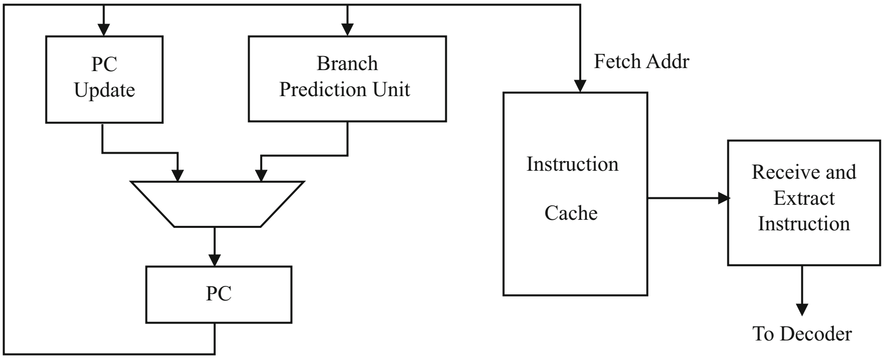


Fig. 3.9 Instruction fetch unit. The PC register is updated either by the branch-prediction unit or the PC update unit. The value of PC is used for fetching the cache line from instruction cache. The instructions are extracted from the cache line and sent to decode unit

Each cache line could contain more than one instruction in a byte aligned organization, with the instruction length being fixed or variable depending on the ISA of the processor. In case of variable length instructions, based on the type of instruction, its length is calculated to mark the start and end bytes in the cache line. The extracted instructions are then sent to the instruction decoder.

3.1.2.2 Decode and Dispatch

In general, an instruction could specify complex functionality that is not directly supported by the available execution units. Hence, the instruction decode logic breaks such complex instructions into a sequence of simpler micro-operations supported by the execution units implementing the same functionality. As shown in Fig. 3.10, the decode logic consists of several parallel decode blocks for different types of instructions. Complex instructions are decoded using a microcode sequencer which internally uses a Read Only Memory (ROM) to generate sequences of micro-operations. The micro-operations specify the source and destination registers along with the operation to be performed.

The dispatch logic analyzes dependencies among the available sequence of instructions and attempts to resolve false dependencies through register renaming. To avoid stalls due to register dependency, an additional set of internal registers is maintained. The dispatcher uses an alias table to rename the source and destination registers in an instruction to avoid stalls due to false dependencies (WAR and WAW).

The instructions are dispatched in program order to the issue queue. The dispatch logic reads the available operand values either from the register file or the reorder buffer. If the operand value is dependent on a previous in-flight instruction which has not completed, then the address of the operand source is stored in place of its value. Appropriate flags are set in the issue queue to represent ready and waiting operands. For every instruction dispatched, a free entry in the reorder buffer is

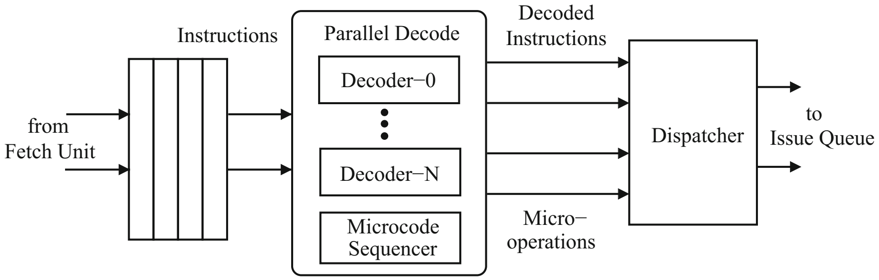


Fig. 3.10 Instruction decode and dispatch unit. The decoder unit consists of parallel decoder blocks. Depending on the type of instruction, they are sent to the respective decoder block. The decoded instruction (series of micro-operations) is then dispatched to the issue queue

reserved. The dispatch logic stalls if free slots are not available in the issue queue or if the reorder buffer is full. After execution of an instruction, the result is stored in the reserved reorder buffer entry. This helps in sequencing the completed instructions in program order.

3.1.2.3 Issue

In this stage, a set of ready instructions in the issue queue is sent to the execution units for computation. An instruction is qualified to be ready if all its operand values are available. Some instructions would be waiting for operand values generated by previous in-flight instructions. The *result bus* is monitored by all the instructions in the issue queue. If the generated result is a source operand for any of the waiting instructions, the value on the result bus is immediately buffered. Hence, instructions in the issue queue become ready as soon as the operands are available. Thus, by waiting for operands dependent on previous instructions, RAW hazards are avoided.

In any cycle, the ready instructions having their corresponding execution unit free are considered for issue (execution). If more than one instruction is competing for a single execution unit, one among them is chosen. Hence, the instructions are issued in an out-of-order manner.

3.1.2.4 Execute

The execution stage performs the computation. Superscalar processors use multiple execution units, allowing parallel computation of independent instructions. The time taken for computation depends on the hardware complexity of the execution unit. For example, floating point units have longer latencies compared to integer units. Also, some of the execution units could be pipelined. Hence, due to the varying latencies of the execution units, the order of completion may not match the order of issue. The results generated by the execution units are forwarded to the issue queue via the result bus. The result value is stored in the reorder buffer slot allocated during the dispatch stage. *Store* instructions directly write the value into the data cache memory.

3.1.2.5 Commit

In this stage the result from the head of the Reorder Buffer (ROB) is written to the architectural register file (ARF). ROB is used to re-arrange the instructions back into the program order. ROB is a circular FIFO structure which commits instructions to ARF only from the head of the queue. That is, an instruction is not committed until all the instructions dispatched prior to it have committed. Thus, ROB stores all results until the previous instructions are completed. This ensures that instructions commit in program order even though they are issued and executed out-of-order.

3.1.3 Power Dissipation

With power dissipation becoming a major design concern for building processors in the recent times, a large body of work has concentrated on building power-efficient hardware structures in the processor datapath. Before discussing the power optimization techniques for the datapath components, we study the power distribution among them to understand the relative importance of operations in terms of their power dissipation and to identify the power hungry components.

Several studies were conducted to understand how the power is distributed within a processor [19, 31, 32, 42]. The power break-up can be studied from various perspectives. Different classifications are possible by studying power division among:

- Attributes of processing: Computation, Storage, and Communication
- Functional Level Operations: Various pipeline stages (Fetch, Decode, Issue, and Commit)
- Hardware Blocks: Datapath Structures (Fetch logic, Decode logic, Issue Queue, Register File, Execution Units, Reorder Buffer, etc.)

Consider the power distribution in Fig. 3.11 of a high performance processor [42]. This distribution shows that the clock is the main power dissipation source, consuming almost 36% of the total processor's power. In this case, the clock power includes the power dissipated by clock drivers, its distribution network, latches, and capacitive loading due to all clocked units. Of all these, capacitive loading is the largest component. The datapath power is contributed by the dynamic logic used for building datapath units. Power dissipated by on-chip storage structures constitutes the memory power. The rest of the power is dissipated by control and I/O interfaces.

The capacitive loading on the clock network is presented by various clocked hardware units in the processor, including various datapath components and registers. So, attributing the power dissipated at the clock interface to clock power, or

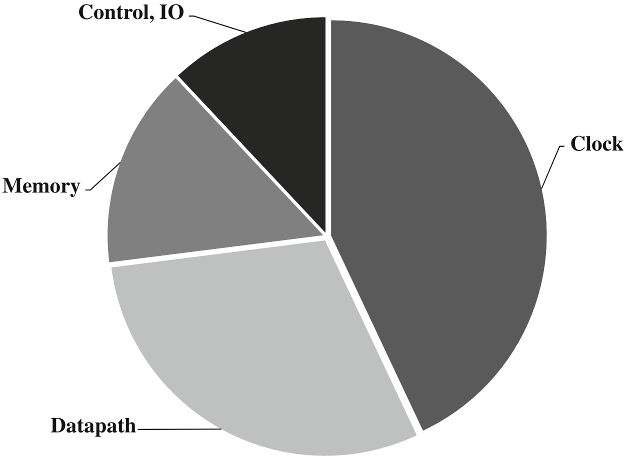


Fig. 3.11 Power dissipation in a high performance CPU [42]: break-up of power dissipation among various attributes (compute, storage, and communication) of processing

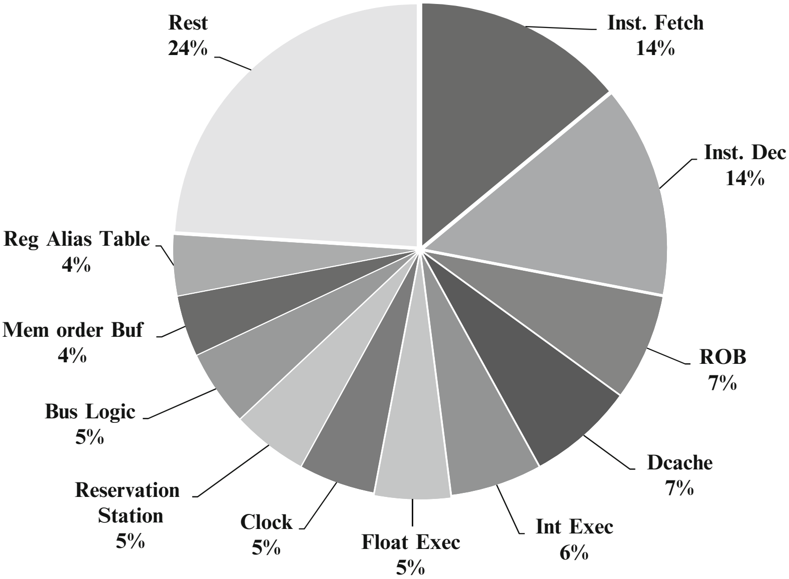


Fig. 3.12 Power dissipation in a PentiumPro chip [32]: break-up of power dissipation among various datapath components in a high performance processor

to the power dissipated by the hardware component, is a matter of perspective. We consider the power associated with capacitive loading of a component to be part of the component power dissipation.

Figure 3.12 shows the break-up of the total power consumed in by an Intel PentiumPro processor. We observe that the instruction fetch, decode, and execution units, along with other datapath components used for storage or support of dynamic scheduling and out-of-order execution, constitute the bulk of the total power dissipated in the chip. Hence, a substantial fraction of the total power is spent in the datapath of a high-performance superscalar processor having complex circuitry for supporting its sophisticated features.

The rest of this chapter is focused on architectural techniques used for building power-efficient datapath components in a modern processor. Each section describes a major datapath component. The functionality of the component is briefly explained to understand its complexity and the source of power consumption in it. Various architectural techniques for reducing power consumption in each of the components and their effect on overall system performance are discussed qualitatively. The power consumed in the memory subsystem (including cache memories) is also substantial, and is addressed in detail in Chapter 4.

3.2 Front-end: Fetch and Decode Logic

The front-end pipeline stages of a processor are responsible for fetching a stream of instructions and decoding them before issuing to the execution or back-end stages. With increasing demand on performance, the number of (issue) ways in a superscalar processor has increased to extract more and more parallelism from the instruction window. Hence, the hardware logic to support such high degree of parallelism has become more complex and power hungry. A significant amount of power is dissipated by the front-end logic in modern superscalar processors, accounting for more than 20% of the overall power consumed by the processor core. Many microarchitectural techniques have been devised to reduce the power consumed by the front-end logic with minimal impact on the overall performance.

3.2.1 Fetch Gating

In modern processors, speculative execution is extensively used to extract useful work during the time that would be otherwise wasted in stalls induced by control instructions. The instruction stream is fetched and executed without waiting for the completion of prior control instructions. Once the branch is executed, the processor verifies the speculated decision and takes corrective action for incorrect predictions. Due to incorrect predictions, all the fetched instructions may not actually proceed for execution. In some cases such instructions could be as high as 80% [4]. Also, due to varying workload conditions, processors seldom work at peak performance. Hence, for most of the time, the rate at which instructions enter the pipeline's front-end does not match the completion rate. Usually, most of the instructions are squashed due to incorrect control flow, leading to improper utilization of pipeline structures.

Though squashing mis-speculated instructions does not affect performance, the energy spent on fetching and decoding them is wasteful. With deep pipelines and considerable amount of energy required for front-end processing, there is a need to save the energy wasted in processing unwanted instructions to improve the performance per watt metric.

The aim of *fetch gating* is to maintain the right number of instructions in the front-end pipeline queue that matches the workload requirement in a given phase of the application [32]. Fetch gating is applied based on the feedback derived by monitoring the state of the application and resource utilization. The information is used to block the instructions from entering the pipeline when conditions are not favorable for their completion. The following techniques are devised to detect conditions favorable for fetch gating.

3.2.1.1 Branch Confidence Estimation

Though modern branch predictors have become increasingly accurate, their aggressive usage results in many instructions that are issued but never committed. To avoid the power wasted in processing instructions that do not commit, the *branch confidence estimation* method uses a confidence estimation strategy to decide when to gate the fetch unit in order to reduce the inflow of instructions [32].

By reducing the number of instructions fetched, decoded, issued, and executed, fetch gating reduces power dissipated in all stages of the pipeline including structures such as issue queue that support dynamic execution. Since performance is the key objective of dynamic superscalar processors, any loss in performance due to fetch gating must be well compensated with the power savings to make it an attractive solution.

Gating logic is an inexpensive hardware attached to the processor pipeline as shown in Fig. 3.13. This hardware unit takes a decision to prevent the fetch unit from bringing new instructions into the pipeline. The decision is based on the confidence estimation information from the branch instructions. The instruction type is

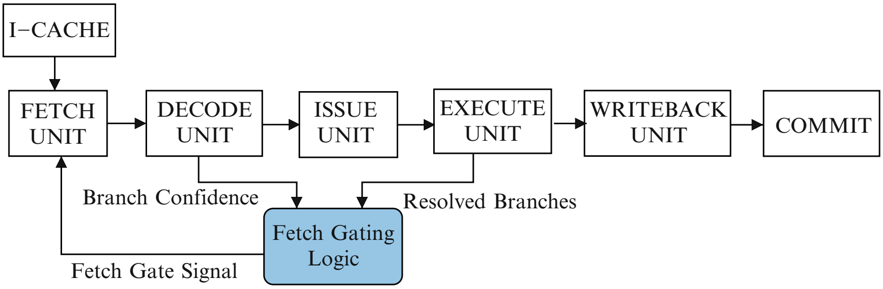


Fig. 3.13 Fetch gating through branch confidence estimation: based on the number of in-flight branches and their confidence estimate, the gating logic controls the throughput of the front-end pipeline stages by stalling them

not known until the decode stage of the pipeline, where the confidence estimation of the branch is calculated for speculative execution. The actual resolution of a branch (or conditional branch) instruction occurs after the execution stage. The confidence estimation information from decode stage and the resolved branch information from execution stage is shared with the newly added decision logic. This information is used to derive the confidence measure (probability of the instruction being committed) for the instructions entering the pipeline.

When aggressive speculation is employed, the probability of a fetched instruction actually committing reduces with increasing depth of speculation (nesting level of control blocks). With the information on the quality of the branch prediction made and the number of such branch instructions in flight, it is possible for the gating logic to calculate the probability of instructions getting committed. If the probability of completion reduces beyond a threshold, then the fetch unit is gated to stall fetching further instructions.

Parameters such as the number of outstanding branches and their confidence measure are considered while generating the fetch gating signal. Also, it is important to study which unit to gate and for how long. Gating can be applied at fetch, decode, or issue units. Usually, maximal benefits can be obtained by employing gating at the fetch stage, as the effect percolates further down the pipeline and gives maximum energy savings. The time for which gating is applied is determined by the gating logic. For example, in Fig. 3.13, the gating logic keeps track of the number of low confidence branches in flight; if the number exceeds a certain threshold then the gating signal is activated and no new instructions are fetched until a certain number of low confidence branches are resolved. The latter is a feedback from the execution stage. Hence, the time for which the fetch unit is gated varies depending on the phase of the application. It is observed that such a scheme would stall the pipeline stage for 2-4 processor cycles indicating a fine control over the speculation mechanism [32].

Confidence estimates determine the quality of the branch prediction. The two important metrics considered for confidence estimates are *specificity (SPEC)* and *predictive value of a negative test (PVN)*. SPEC is the fraction of all mispredicted branches detected as low-confidence decisions by the estimator. PVN is the probability of mispredicting a low-confidence branch. A larger SPEC implies that most mispredicted branches have low-confidence. Large PVN implies that the low-confidence branch is likely to be mispredicted. Both these parameters are important to rate a confidence estimator. The aggregate value of N low confidence branches is considered while taking a gating decision to avoid stalling the pipeline too frequently and thus minimizing performance degradation.

The effect of gating on performance could be positive or negative depending on the nature of the workload. In some cases speculative execution helps warming up the caches, but due to fetch gating, the benefits of warm-up effects are lost resulting in loss of performance. In other situations, fetch gating actually helps improving performance by blocking some of the incorrect path instructions from occupying the pipeline resources, thus allowing the correct path instructions to complete faster.

3.2.1.2 Rate Mismatch Flow Control

It is often observed that several applications do not exhibit the large degree of parallelism for which the processor is designed, or at least do not do so during many phases of program execution. In these situations, though the processor works at full throttle, it does not improve the performance significantly. Detecting such conditions and reducing the instruction fetch/decode rate would lead to significant power savings without affecting performance.

In this method, front-end gating is done based on instruction flow prediction which is oblivious to the confidence metric discussed in Section 3.2.1.1 [4]. Consider the example illustrated in Fig. 3.14, where a 4-way issue processor has to execute four instructions A, B, C, and D in program order. Instruction D is dependent on the result of instruction C. In general, a processor cannot fetch instructions in two cache blocks in a single cycle. Hence, assuming that the instructions A, B, C, and D belong to two different cache blocks, it takes two clock cycles to fetch the four instructions. In normal execution flow, i.e., when the decode stage is enabled every cycle, instruction A is decoded in cycle-2 while the instructions B, C, and D

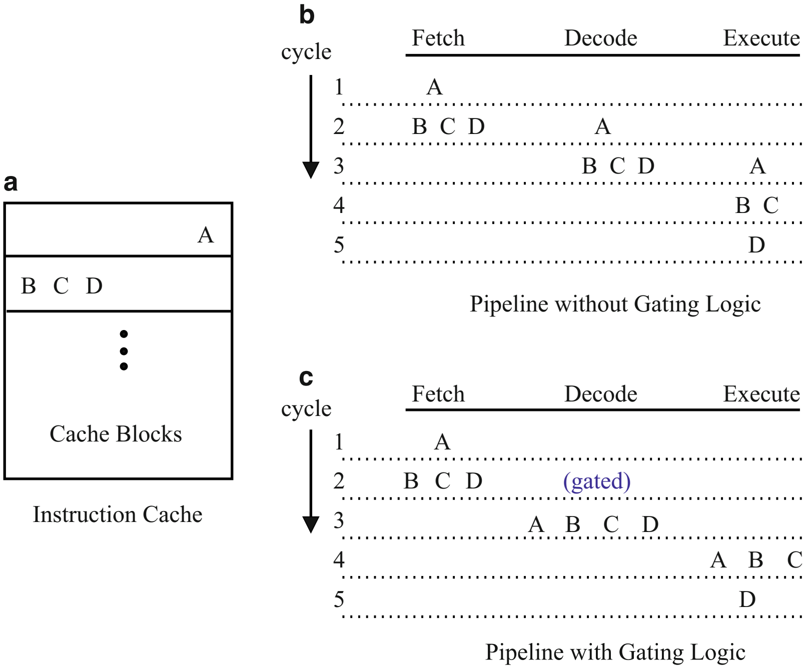


Fig. 3.14 Front-end gating for power efficient usage of decoder. When peak throughput is not expected or when the back-end stages of processors are working at a slower rate, gating of the front-end stages could save power without affecting performance. Instruction cache content is shown in (**a**). Compared to the normal execution flow in (**b**), the gating technique in (**c**) saves power by stalling the decoder in cycle 2 without affecting performance

are decoded in cycle-3. Since instruction D can execute only after execution of C, it takes a total of five cycles to execute all the four instructions. During these five cycles, all pipeline units are active and dissipate power even when they are not working at full capacity. By employing instruction flow-rate feedback to control/gate the decode units, it is possible to switch off the decode stage without losing any performance. In the above example, by switching off the decode stage until all four instructions are available for decode, i.e. during cycle-2, the processor decodes all four instructions during cycle-3 and executes independent instructions A, B, and C in cycle-4. It delays the execution of instruction D, which is dependent on result of C, to cycle-5. Thus, this scheme could provide a low power solution by gating front-end stages when performance is not affected or when the pipeline is not operated at full throttle.

Flow based gating would require monitoring various pipeline stages to estimate the required degree of parallelism. Gating is applied if the required parallelism is expected to be attained even after stalling the fetch and/or decode stage. The gating can be applied on a cycle-by-cycle basis; hence the control is very fine. Parameters that could be observed to monitor the required parallelism during a program execution are:

Decode and Commit Rate: If the decode rate in the processor is higher than the commit rate, it implies that the extraneous work being done is due to either the incorrect path instructions or due to little parallelism available for execution. In such conditions the decode unit can be stalled with minimal performance degradation.

Dependence Information: If the dependence information is kept track of after the decode stage, then it is possible to identify the number of independent or parallel instructions ready for execution, which helps calculate the number of parallel decoders required to support the instruction stream. If the dependency between the instructions is high then the average parallelism required would be low. This parameter can be used to stall the front-end stages while still maintaining the maximum instruction flow.

Thus, considerable power savings can be achieved using instruction flow based front-end gating. We discussed techniques to identify favorable conditions which are oblivious to the confidence metrics discussed earlier. It is also possible to combine the instruction based methods with confidence based gating to devise a more power efficient front-end gating technique.

3.2.2 Auxiliary Decode Buffer

In general, applications spend most of their execution time in instructions within loop nests. This observation holds for a large number of application domains, including signal processing, multimedia, and other scientific applications.

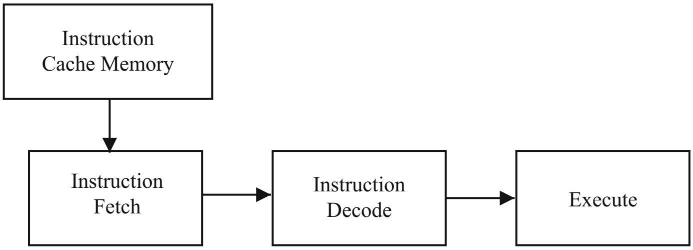


Fig. 3.15 Conventional processor pipeline

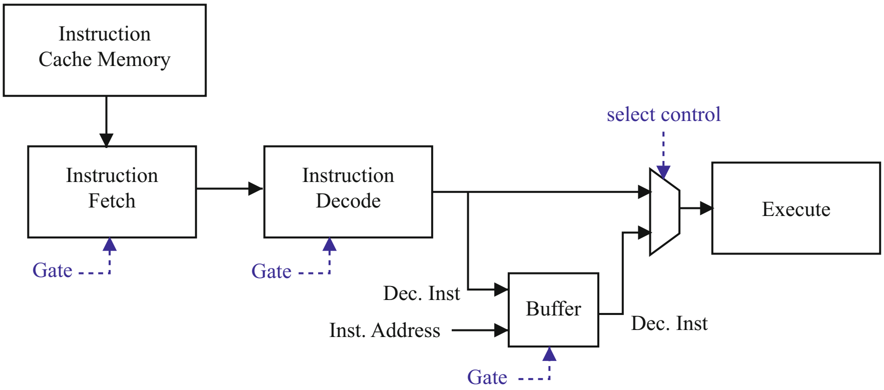


Fig. 3.16 Pipeline with auxiliary Decode Instruction Buffer (DIB) [21]. An additional buffer is inserted between the decode and execution stages to store decoded instructions. While executing loops, the buffer is filled with decoded instructions during the first loop iteration. For subsequent loop iterations, decoded instructions are read from the buffer. This allows us to gate the front-end stages from the start of second iteration to the end of loop execution, thereby saving power

In a conventional processor pipeline, shown in Fig. 3.15, the fetch unit reads instructions from instruction cache memory, which are then processed by the decoder logic to generate a sequence of micro-operations. The execution units perform the computation. During the execution of loops, instructions in the loop body are repeatedly executed a large number of times. Though temporal locality and availability of on-chip caches ensures no performance loss while fetching the instructions (i.e. no cache misses), a large amount of power is wasted by the front-end pipeline stages in repeatedly fetching and decoding the same set of instructions.

Front-end power dissipation can be reduced for loop kernels by an auxiliary buffer to store decoded instructions [21]. The execution pipeline is modified by adding a small buffer after the decode stage as shown in Fig. 3.16. The buffer is addressed by the program counter and is used only for loop kernels; in all other cases the buffer is gated and instructions flow in normal manner from decoder to the execution units of function units (FUs).

During the first iteration of the loop execution, the buffer is filled with the decoded instructions of the loop body. For the remaining iterations of the loop, the decoded instructions are read from the buffer instead of being fetched and decoded again. The fetch and decode units are gated after the first iteration of a loop. The size of the auxiliary buffer required for real applications is usually small, leading to a negligible power overhead compared to the power consumed by the fetch and decode logic. Since most of the computationally intensive applications spend more than 80% of their execution time in loops, the pipeline architecture with the auxiliary buffer will save a substantial amount of power in the front-end stages.

3.3 Issue Queue / Dispatch Buffer

Dynamic scheduling in an out-of-order superscalar processor would require many supporting hardware units. The Issue Queue or Dispatch Buffer is one such structure which holds the decoded instructions waiting to be executed. It also helps in removing false dependencies through register renaming. A black-box view of the issue queue is shown in Fig. 3.17(a). The instructions dispatched by the decoder along with the required operands (if available) or their source (if generated by a previous instruction in flight) are stored in the issue queue. Thus an instruction in the issue queue could be in one of three states:

- ready for execution, or
- waiting for an operand (to be generated by a previous instruction), or
- waiting for a function unit (which is busy executing any other instruction).

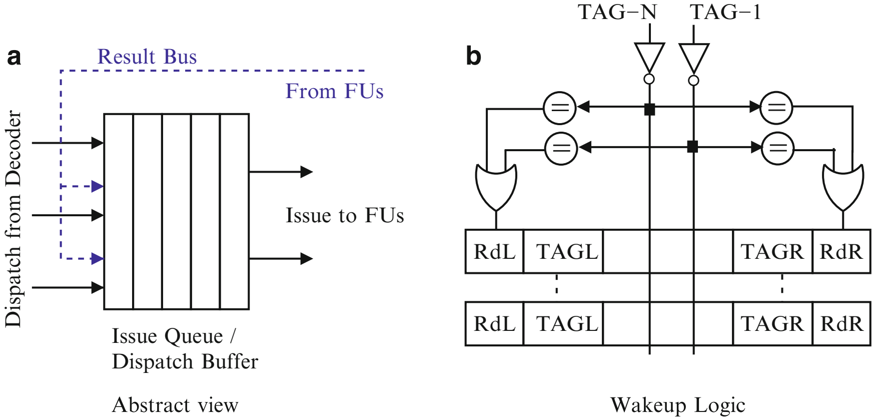


Fig. 3.17 Abstract view of issue queue and wake-up logic. (**a**) Dispatched instructions are stored in the issue queue. The tags from the result bus are broadcast to the issue queue. (**b**) Broadcast tags are compared with all issue queue entries and a match leads to buffering of the corresponding operand. The issue logic sends one or more ready operations to the available function units for execution

The result of execution and its associated tag is forwarded (broadcast) to all instructions in the issue queue through the result bus. The issue queue performs an associative search operation to locate instructions waiting for this operand by comparing the broadcast tags (*TAG-1* to *TAG-N*) with tags of source operands (*TAGL* and *TAGR*). The entries with matched tags will store the broadcast result and sets their ready flags (*RDL* and *RDR*). When all input operands are available (i.e., both *RDL* and *RDR* are set) for an instruction in the issue queue, it becomes ready to execute.

The issue queue can be implemented as a single structure holding all instructions or split into multiple issue queues, also called reservation stations, where each queue is attached to a function unit (or group of FUs). When a reservation station is attached to a single function unit, its scheduler has to choose only one ready instruction in a cycle. In case of shared queues a more complicated scheduler is required to select multiple ready instructions. A centralized queue, though flexible, requires an even more complex scheduler. In hardware, the issue queue is implemented using one of the following structures:

Circular Buffer: In this architecture, shown in Fig. 3.18, the newly dispatched instructions are added at the tail of the circular buffer. Instructions close to the head pointer are older and have a higher priority during selection (for issue) among the ready instructions. Thus, this architecture inherently maintains the relative age among the instructions leading to simpler selection hardware. However, when instructions are issued in an out-of-order manner, it results in holes/gaps between tail and head pointers leading to inefficient buffer utilization. A compaction scheme can be designed for better utilization of the queue at the cost of additional hardware complexity.

Random Buffer: In this architecture, shown in Fig. 3.19, the buffer is implemented using a RAM structure. This allows the dispatched instructions to be written in any free entry, leading to efficient buffer utilization without requiring additional

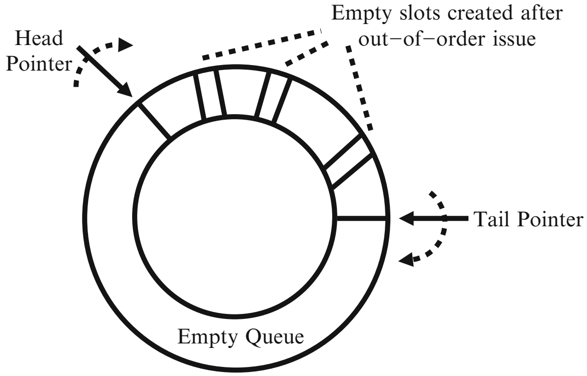


Fig. 3.18 Logical view of the issue queue implemented as a circular buffer. The tail pointer follows the newly issued instructions, while the head pointer points to the oldest instruction in the queue. Out-of-order issue results in "gaps" between the head and tail pointers

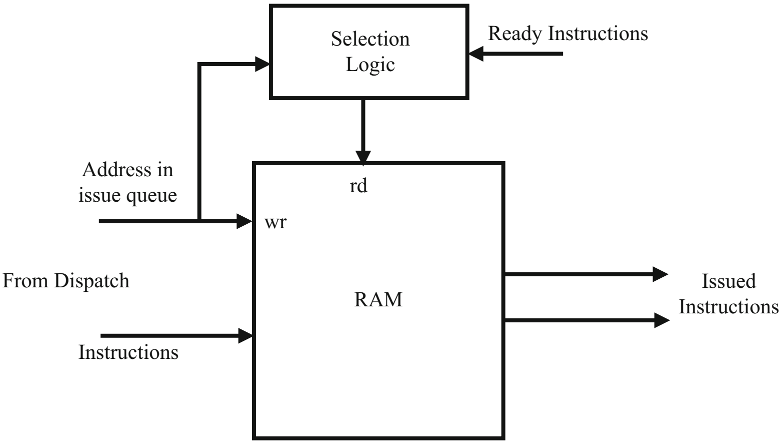


Fig. 3.19 Logical view of the issue queue implemented as a random buffer. Dispatched instructions can be stored at any location in the RAM. Based on the priority, the selection logic issues a set of instructions among the ready ones

hardware (such as compaction logic). The selection logic tracks the relative age among the instructions in the buffer and based on this information it selects the instructions (among the ready ones) for issue. As it is harder to maintain the relative age among the instructions in a random buffer, the selection logic becomes more complex in this architecture.

Among the structures supporting dynamic scheduling, the issue queue is one of the major power dissipation sources in the processor. Power is dissipated in the issue queue during the following operations:

- When an instruction is dispatched, energy is dissipated in locating and writing into a free entry of the issue queue.
- When the execution is completed, the result value along with its tag address is forwarded to all issue queue entries via the result bus. During the associative search for matching the forwarded data, the tag comparators are activated for all queue entries; this consume a significant amount of power.
- Power dissipated during issue is spent in arbitrating for a function unit, enabling a ready instruction, and reading its operands from the queue before sending them for execution.

Figure 3.20 shows the power distribution within the issue queue. The issue operation consumes almost 50% of the total power dissipated by the queue, while dispatch and forwarding operations consume about 20–25% of the total queue power. Now that we have established the issue queue as being one of the most power hungry structures in the datapath, the rest of this section describes various techniques to reduce its power consumption while minimizing the impact on performance.

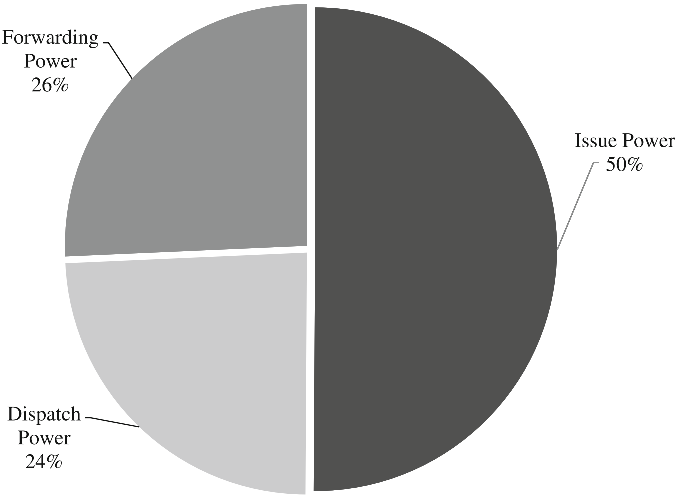


Fig. 3.20 Break-up of power dissipation in the issue queue [27]

3.3.1 Dynamic Adaptation of Issue Queue Size

A significant fraction of the energy in the issue queue is spent on the associative search and wake-up logic. It is observed that up to 60% of the energy in the issue queue is spent on such operations [16]. In every cycle, all the entries in the queue are compared with the broadcast tags to buffer the result for waiting operands. However, such operations are unnecessary for entries that are either ready for execution (all operands available) or empty (invalid entries).

The size of a queue has the maximum effect on the power it dissipates; the required size of the issue queue varies from application to application and between various phases within an application. Hence, tuning the issue queue size dynamically depending on the instruction stream could achieve considerable power savings.

Different entries in the issue queue are very similar in terms of their contribution towards energy consumed, but are dissimilar when their contribution to performance is considered. As discussed earlier, wake-up logic consumes equal energy for empty entries, ready entries, and the entries that are actually waiting for operands. It is observed that empty entries contribute up to 70% of the total wake-up logic energy and the ready entries contribute another 15% of the energy. On the whole, around 85% of the energy consumed by the wake-up logic is spent on useless operations which do not contribute to performance. Two techniques used for curtailing the energy wasted in the wake-up logic are as follows.

Disable Wake-up for Empty Entries: The CAM cell structure of the queue is modified to provide a gate/switch for disabling tag comparison logic. In each cycle, the match line is precharged and conditionally discharged based on tag contents.

The precharge line is gated to save the dynamic power dissipated during tag matching (wake-up). The gating is applied based on a bit that indicates whether the issue queue entry is valid. Such gating would prevent all the empty or invalid entries in the queue from dissipating energy for tag comparison.

Disable Wake-up for Ready Entries: Once ready, an entry remains in wake-up state until it is sent for execution, thereby wasting energy for all ready entries. As in the case of empty entries, the precharge line can be gated to avoid dynamic power dissipation for wake-up logic. The gating signal in this case would be a bit indicating the availability of all operands, i.e., a ready entry. The energy consumed by the additional logic itself is negligible when compared to energy savings achieved.

By employing the above two methods, substantial (upto 90%) energy wastage in the wake-up logic can be avoided at the cost of increased area in terms of additional logic used for gating the precharge lines in all entries of the issue queue.

In general, processors are designed with a fixed issue queue size targeting maximum performance. However in many cases, depending on the type application or phase of application, each part within the issue queue has a different contribution to performance. For a FIFO based issue queue, it is observed that, in a program exhibiting maximum parallelism, the youngest (or most recently filled) part of the issue queue has negligible contribution towards the IPC. The reason is simple: most of the required parallelism is supplied by the older (higher priority) instructions in the issue queue. Similarly, when the dependency between the instructions is large or a minimum parallelism exists among the available instructions, it is very unlikely that the younger entries would contribute towards the IPC. Hence, there are time periods during which the younger part of the issue queue would have negligible contribution towards performance but still consumes significant amount of power.

To exploit this condition for further power savings in the issue queue, a dynamic scheme is devised which monitors the contribution of younger entries towards the IPC or performance. If their contribution is found negligible then the size of the issue queue is reduced and it is restricted to operate with the number of entries that actually contribute towards performance. Since the dynamic decision is taken based on performance, it has negligible effect on overall performance while saving power whenever possible by dynamic adaptation of the issue queue size. In this approach, the issue queue is sized to match its level of utilization or to the necessary instruction window demanded by the application [8, 16]. Thus, unnecessary entries are shut down, saving considerable energy.

3.3.2 Zero Byte Encoding

The *zero byte encoding* power optimization technique exploits the fact that, during the execution of any real application, significant bytes in many of the operands are all zeros. For example, during the execution of SPEC-95 benchmarks on a cycle level simulator, it was observed that, on an average, more than half of the bytes in the operands were zeros [18]. This is due to the fact that many operands in a real program are either small literals, byte level operations, or address offsets. Hence during execution, when such instructions flow through various pipeline stages of a processor, a considerable power is wasted in storing, processing, and driving the zero bits of the operands. This accounts for a significant amount of overall power.

Zero-Byte Encoding is a power optimization technique proposed to avoid such wastage of energy on processing the zero bytes. This technique utilizes extra bits to encode the zero bytes in an operand to dynamically switch-off the function units and memory structures to avoid transmission, processing, and storage of unwanted zeroes [6,18]. This technique, which is generally used for reducing power consumption in caches and function units, can also be applied to the issue queue [27].

To implement this in hardware, an additional zero-indicator bit is stored with each byte of the word (operand), indicating an all zero byte. During dispatch, this bit is used to disable the writing of zero bytes into the dispatch buffer (issue queue). Hence, energy consumption is reduced by driving fewer bit-lines. Similarly during issue-stage, the zero-indicator bit is used to gate the word select signals to disable the reading of zero bytes. This technique significantly reduces the wasteful energy spent in reading and writing the zero-bytes in the issue queue structure. The energy savings are at the cost of increased (around 11% [27]) silicon area of the issue queue for storing the zero-indicator bits and for the logic introduced for dynamic gating of the hardware units.

3.3.3 Banking and Bit-line Segmentation

The issue queue is essentially a register file except that it requires associative searches during a write operation. Writing to the issue queue involves a search operation to locate a free entry in the queue. Since the queue structure is similar to that of a register file or a RAM, some of the energy saving schemes used for them are also applicable for the issue queue. Banking and Bit-line segmentation [17] are a few such techniques used for reducing energy consumption of the issue queue [27].

The number of rows in the queue has a linear effect on the bit-line capacitance. Thus, large issue queues would require higher energy per access. Bit-line segmentation and banking reduces the capacitive loading by splitting the bit-lines into multiple segments. As shown in Fig. 3.21, the issue queue is split into multiple CAMs and RAMs. CAM cells store tags for associative look-ups while the less power-hungry RAM structures are used to store the instruction data. Each bank of the CAM and RAM can be turned off independently when an access to the bank is not required, thereby requiring lower energy per access. Chapter 4 explains these techniques in more detail for the memory structures.

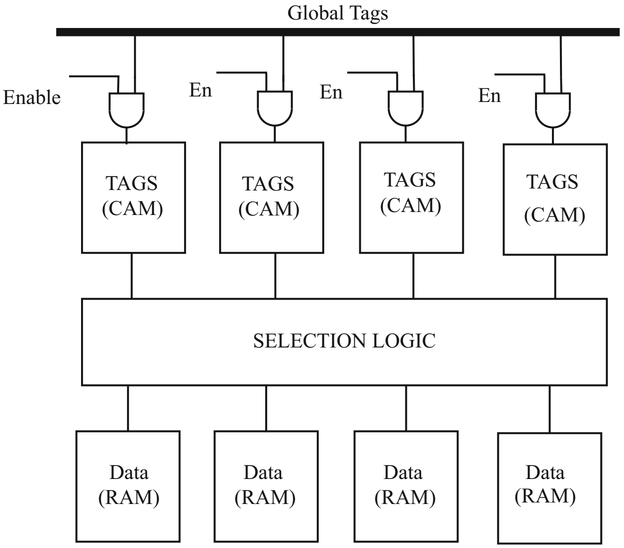


Fig. 3.21 Banked issue queue. Bit line segmentation and banking reduces capacitive loading, thereby reducing the energy per access. Banking also provides an opportunity for turning-off certain banks when they are not used

3.3.4 Fast Comparators

From the simulation of SPEC-95 benchmarks, it is observed that, on an average, less than 20% slots in a dispatch buffer will be actually waiting for the data in any given cycle [27]. This property is exploited for saving energy, as described in Section 3.3.1, by disabling undesired entries for comparison. Further, in any given clock cycle, of the remaining 20% of slots actually waiting for data, only a few will match the broadcast tags; a majority of them would result in a mismatch. In any cycle, on an average, only 2-4 comparisons result in matches while the rest result in a mismatch.

Generally, the associative searches or comparisons are done using the typical pull-down comparators for a fast response time. These comparators dissipate more energy on a mismatch than for a match. Hence, using such comparators in an issue queue is power-inefficient because most matches are unsuccessful. Therefore, a comparator circuit that dissipates negligible energy on a mismatch is used in the associative search logic for issue queues.

3.4 Register File

Register files are used to store most of the operands required by the instructions in a processor. Register file size directly determines the number of instructions in-flight by stalling the dispatch stage if the free registers are exhausted. The increasing demand on parallelism resulted in designing wide-issue processors with large instruction windows and support for multi-threading. Supporting a large number of instructions in-flight would require a correspondingly large number of operands to be stored in the register file. Also, wide-issue processors resulted in multi-ported register files for supporting simultaneous access. For example, a 4-issue processor would require 8-read ports and 4-write ports. Thus, the increasing size and ports of a register file makes it one of the major power consuming structures in the datapath. In modern processors up to 15% of energy is consumed by the register file, making it one of the hot-spots in a processor.

Increasing size and longer access times of power hungry register files are tackled through techniques that reduce the effective register file size and/or number of ports. This section describes some popular techniques used for designing low power register files.

3.4.1 Port Reduction and Banking

Conventionally register files are implemented as many-ported RAM structures. An eight-issue processor would theoretically require a register file, shown in Fig. 3.22, with 16-read ports and 8-write ports for a single cycle operation.

Increasing bandwidth requirements for supporting higher frequencies and multi-threading makes it difficult for single cycle implementation of register files. Also, the multi-cycle implementations require complex logic degrading the performance of a processor. Thus, more scalable techniques and structure organizations, similar to data caches, are used to reduce the power consumption.

3.4.1.1 Reducing Port Requirements

Increasing the number of ports increases the complexity of register file memory cell and interconnections, which ultimately leads to increased power dissipation. The

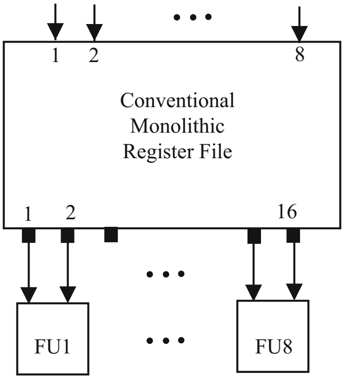


Fig. 3.22 Conventional Register File Architectures. A multi-ported register file with N-write and $2N$-read ports. Each read port is connected to a specific function unit

following observations and techniques are utilized to reduce the port requirements of a register file:

Bypass Network: With the availability of the bypass network and with instructions that either require less than two operands or do not write results to the register file, the average number of ports required per cycle is, in practice, less than the theoretical value. It is observed that more than 50% of the operands can be fetched through the pipeline's bypass network, relieving the pressure on the required number of ports [3]. Hence, replacing a 24-ported register file with 8-ported register file, shown in Fig. 3.23, for an 8-issue processor results in only a small (around 2%) degradation of IPC (due to port conflicts), but provides significant energy savings and better access times compared to the conventional structure. The energy per access for the 8-port register file is almost 4-times lower when compared to a conventional register file.

Using Auxiliary Buffers: In real applications, most of the results generated by function units are consumed by instructions waiting in the issue queue within a few cycles of their generation. This property can be exploited to reduce the number of register file accesses by either directly forwarding the results from the bypass network or storing the results in a small auxiliary structure for the next few cycles until they are used. Since the size of such an auxiliary queue is very small, its energy per access is much lower compared to that of the register file. Also, this reduces the port requirement of the register file. Reducing the number of ports by using such auxiliary queues could lead to significant energy savings at the cost of a small performance loss [24].

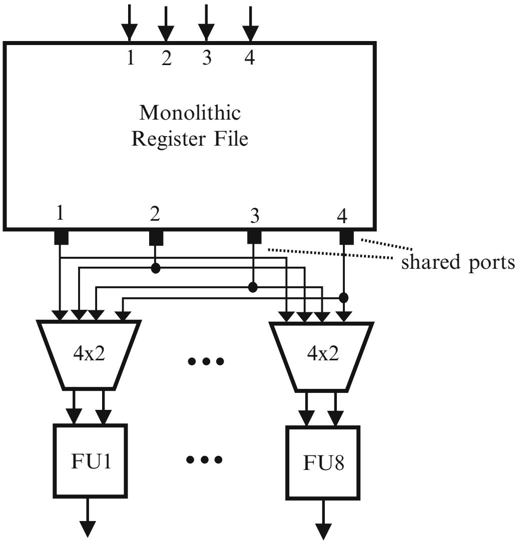


Fig. 3.23 Reduced port register file architecture. By minimizing the number of ports, the energy per access is reduced. Here, one read port is shared between more than one FU. The reduction in register file bandwidth is compensated by the availability of operands in the bypass network and hence results in minimum performance loss

3.4.1.2 Banking

Further savings in energy could be obtained by banking the register file structure as shown in Fig. 3.24. In banked register file architecture, the registers are split into multiple banks. If the registers are split into N-banks and each bank has p-ports, then such an architecture could provide upto $N \times p$ operands in one cycle but with the limitation that no more than p operands can be fetched from a single bank in one cycle. Compared to a single RAM with $N \times p$ ports, the complexity and energy per access of a banked architecture is vastly reduced.

Figure 3.24 shows a register file with 4-banks, each with single read and write ports. In cases where two operands are required from any bank in a cycle, the instruction is stalled for one cycle. Also, when multiple instructions compete for a single bank, only one of them will succeed in fetching the operand. Thus, an additional arbitration stage is required in the pipeline to resolve conflicts when ports/banks of a register file are shared. Empirically, it is found that no more than 5% degradation in IPC occurs due to port/bank conflicts [3]. Port/bank conflicts could be reduced by efficient distribution of operands among the available register banks (e.g., operands of the same instruction assigned to different banks). Banked architectures reduce the bit-line capacitance by partitioning the length of bit-lines into multiple segments. During register access, only the banks that are addressed are activated, along with their corresponding decoding circuitry. On an average, fewer banks are activated per cycle, reducing the overall power consumption. The banked architecture in Fig. 3.24 has almost 4-times lower energy consumption compared to the reduced-port architecture in Fig. 3.23 [3].

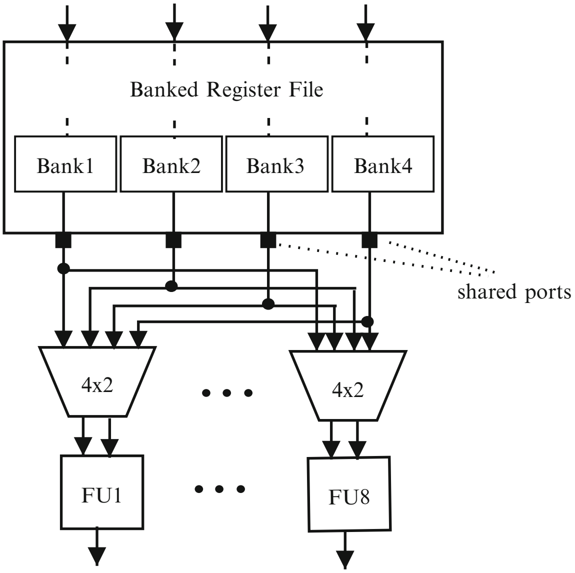


Fig. 3.24 Banked register file architecture. Banking reduces the complexity of a register file and hence the energy required per access. Execution is stalled when more than one data element is required from the same bank in a given cycle. However, its effect on performance can be minimized by efficient register allocation

Efficiency of banking can be improved by exploiting the fact that most of the operands would not occupy the full bit-width (usually 32-bit or 64-bit) provided by the register file. Most of the operands are narrow, as explained in Section 3.3.1, and occupy only a few LSB bits [18]. A 64-bit register file is implemented by partitioning it into two banks, one containing the lower word (LSB bits) and other containing the upper word (MSB bits) [25]. This will avoid reading unnecessary zeroes when narrow operands are accessed and thus improve average energy per access.

3.4.2 Clustered Organization

The access time and energy per access are adversely affected with the increasing size of register files. Thus, for scalability, large register files could be implemented by partitioning the storage and computational units into multiple clusters [15, 35]. Each cluster consists of a register file supplying operands to function units within the cluster. An interconnection network is designed to support inter-cluster communication. Decentralizing the register file structure into smaller and simpler units results in faster access times, lesser number of ports, and reduced energy per access in the register file.

Figure 3.25 shows a two-way clustered architecture [45]. The register file (as well as issue window) is split between the clusters, each feeding a different set of function units. If the operands are available in the local register file, then the renamed instructions are dispatched conventionally into the issue window, otherwise a request is sent to the Remote Access Window (RAW) to fetch the value from a remote register file into the local Remote Access Buffer (RAB) through one of the dedicated ports of the register file. However, accessing remote register files would incur a penalty in terms of higher latency thus reducing the IPC when compared to a processor with a centralized register file architecture. Energy as well as performance efficiency is improved by minimizing inter-cluster communication. Clustered architectures are power-efficient and provide higher performance per watt than centralized architectures.

3.4.3 Hierarchical Organization

The IPC of a machine scales with increasing size of the register file. For example, the IPC of an 8-issue machine would scale with register file size beyond 128 entries [14]. However, it is very difficult to design such large monolithic register file structures due to cycle time and power constraints. Thus, hierarchical register file structures are proposed to eliminate such scalability issues [3, 12, 44].

Figure 3.26 shows a hierarchical register file architecture. It has a multi-level organization with each level having a different size, ports, access times, and

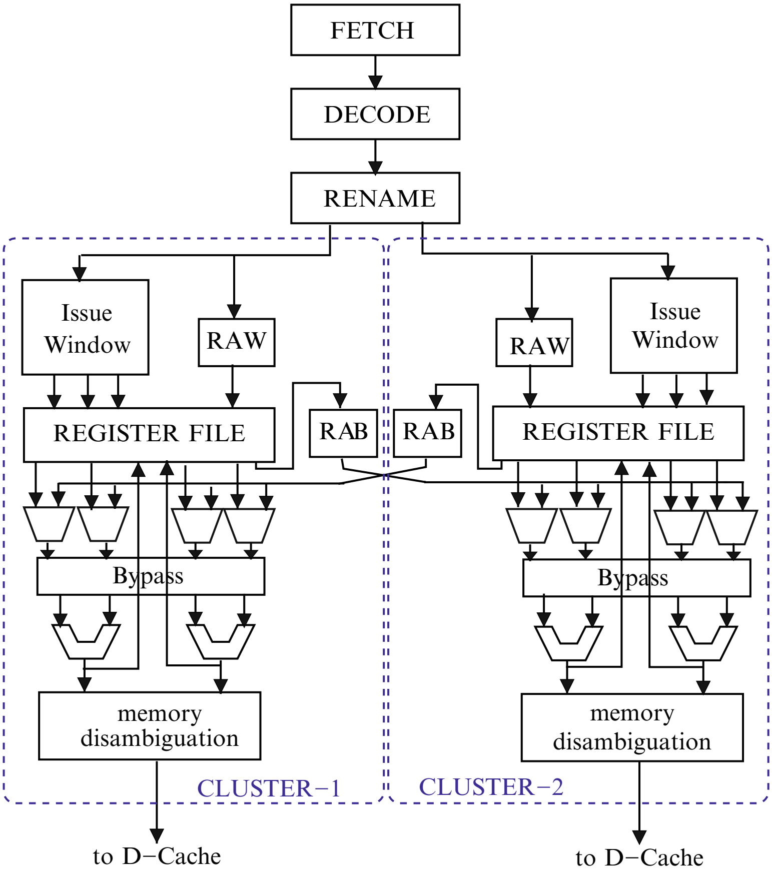


Fig. 3.25 Clustered register file architecture. Splitting the register file into multiple clusters reduces its hardware complexity and results in lower energy per access. However, this increases the latency due to the communication overhead in fetching the register value from a different cluster

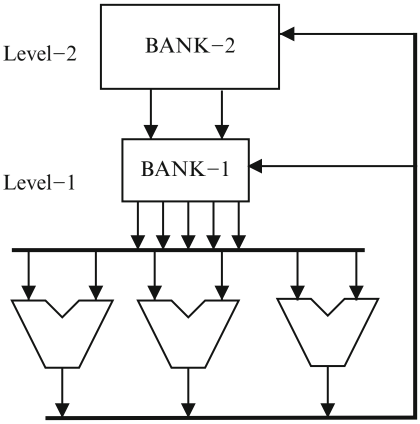


Fig. 3.26 Hierarchical register file (Register Cache) architecture: organizing registers into multiple levels reduces the port requirement on each bank, making it more power efficient

access energy. The bank (at lowest level) closest to function unit directly supplies the operands to function units. Operands stored in upper level banks (away from FUs) are cached into lower level banks. The results from FUs are written into any level depending on its usage in future time. Such a register file architecture, resembling cache hierarchy, is often called *register cache*.

The upper level banks have fewer ports but more number of registers and longer access times. The lower level banks have fewer registers and more number of ports with single-cycle access time. When results are produced, they are stored in upper levels and, in some cases, they are also written to lower levels depending on its predicted usage in near future. Data is never required to be transferred from lower to upper levels as upper levels are always updated with results. A prefetch mechanism is used to transfer data from upper to lower levels of register memory. Such an architecture improves performance by reducing access times and also saves considerable energy by directing most of the accesses to smaller memory banks. Caching and prefetch techniques in a hierarchical register organization are described below.

Caching: Unlike caches, registers do not exhibit much spatial or temporal locality. Moreover, it is observed that almost 85% of register values are read only once during execution of a program [12]. Thus, in case of hierarchical register organization, FU results that are not read from bypass logic are cached at the lowest level.

Prefetching: As in caches, prefetching can be employed to overlap the data transfer time between upper and lower levels of register banks with execution time. This requires knowledge of dependency relations between instructions in flight. The decode stage of the pipeline could determine the dependency information which can be used for the purpose of prefechting. Consider the following sequence of instructions after renaming:

$$\begin{aligned} I1 &: r1 = r2 + r3 \\ I2 &: r4 = r2 + r5 \\ I3 &: r6 = r1 + r8 \end{aligned}$$

When instruction *I1* is issued register *r8* is prefetched to overlap the execution time of *I1* with data transfer time of *r8* from upper to lower level of register banks.

3.5 Execution Units

The execution units consist of dynamic logic circuitry for various integer and floating point arithmetic operations required to support the instruction set of a processor. A large number of arithmetic units are required to aid the high issue width of modern processors. Also, deeply pipelined and/or parallel hardware is required for operating under high clock frequency and hence, a considerable amount of power is consumed by the execution units. This section describes techniques to reduce dynamic and static power consumed by the execution logic.

3.5.1 Clock Gating

Clock gating is a popular technique for reducing dynamic power dissipation. In this method, the clock supplied to a circuit is gated using a control signal which is triggered when the circuit is not in use. This is based on the observation that, not all hardware is used in every cycle, yet power is dissipated due to the constant charging and discharging with clocks pulses. Clock gating such circuits avoids the unnecessary toggling and thus greatly reduces the average dynamic power dissipation. Chapter 2 discusses this technique in detail.

An important aspect of clock gating is to identify when to gate and for what duration to gate. This information is required in advance to take proper gating decisions. Clock gating results in significant energy savings and is the most popular and widely used technique for reducing dynamic power consumption. Some techniques used for gating the processor execution units are discussed here.

Opcode Based: The information regarding the type of instruction is known only after the decode stage. Hence, by the end of the issue stage one can determine which execution units are not going to be used in the next cycle. This information is used to generate gating signals for the unused units [30]. Clock gating signals can also be used to prevent driving the high capacitive result bus when no execution unit is scheduled to generate results in that cycle.

Value Based: Even though the processor data path is designed for 32 or 64-bit, it is observed that in most of the real applications, as many as 50% operations would require only 16-bit or even less, as mentioned earlier. To avoid unnecessary computations on zero values, gating signals are used to activate the function unit for different bit-widths. At run time, based on operand values, the appropriate width of the function unit is activated. Up to 30% energy savings were reported when this technique was applied to integer function units [7].

3.5.2 Operand Isolation/Selective Evaluation

Operand isolation or selective evaluation is a dynamic-power optimization technique that avoids propagation of signals that cause redundant activity in the logic downstream [11,34,41]. It provides fine control by selectively blocking such signals. Blocking is done using additional circuitry that prevents the signals from initiating any redundant operations downstream.

Figure 3.27(a) shows a part of a logic circuit, where inputs A and B are supplied to a function unit FU. The result of the operation is conditionally written to register *REG*. For cycles when the register *REG* is not loaded with the output of FU, any computation by FU is redundant. Computation by FU is caused due to toggling of inputs A or B. Undesired toggling in A and B when the output of FU is redundant results in wastage of power.

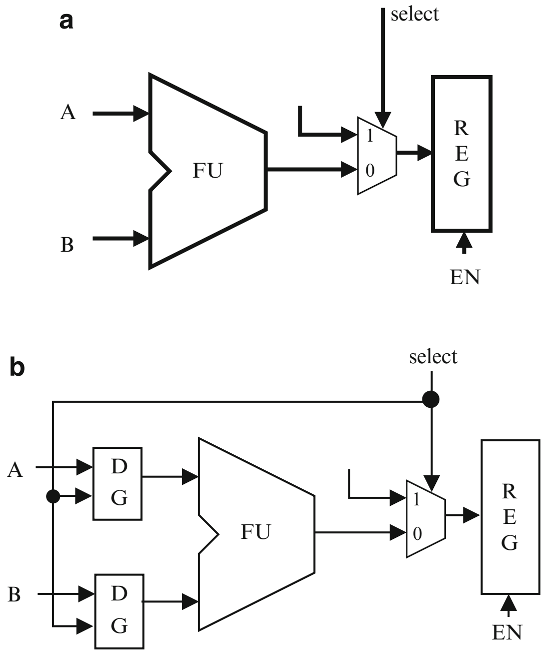


Fig. 3.27 Operand isolation example: when the output of FU is not required to be written to the register (i.e., select='1'), then the inputs to the FU are prevented from toggling, reducing dynamic power dissipation in FU. (**a**) Logic Circuit. (**b**) Logic with Operand Isolation

To avoid redundant computations, the signals/inputs *A* and *B* are blocked from propagating to the *FU* using additional circuitry (such as transparent latch, AND gate, OR gate, etc.), at the inputs of *FU*, as shown in Fig. 3.27(b). Based on a control signal, the inputs are either allowed to propagate, or blocked to retain the previous value. In this example, the *select* signal is used to block *A* and *B* when the result of *FU* is not loaded into *REG*, thus saving power by preventing undesired computations by *FU*. Needless to say, this optimization affects the area and cycle time, but it is worth the savings it provides in terms of energy.

3.5.3 Power Gating and Multi-threshold Logic

Power gating for static or leakage power is analogous to clock gating in case of dynamic power. In this technique, the power source is cut-off, thereby shutting down the supply voltage to a function unit. With no path from source to ground, the leakage current is blocked.

In current technology nodes, leakage current is considerably high in subthreshold regions. Hence, multi-threshold voltage logic is used to reduce leakage power in function units of a processor [29]. Chapter 2 discusses power gating and multi-threshold logic in detail.

The logic is switched to sleep mode during idle periods to minimize leakage power. The cost, in terms of power dissipated and time for transition, of switching between sleep and active modes is considerably high. Thus, frequent switching would degrade the processor performance considerably. Therefore, for both power gating and multi-threshold logic, it is very important to identify in advance, the favorable conditions (when and for what period) under which switching a function unit between power modes is likely to save power with acceptable performance degradation. The micro-architectural and compiler techniques that are used to identify such conditions are discussed below.

3.5.3.1 Time Based

A simple way of predicting idle periods is to monitor the state of a function unit; if a streak of cycles is observed during which the function unit is idle, then it is likely that it will be idle during the next few cycles. Hence, the function unit is put into a low power mode by using any of the two techniques discussed earlier. To implement this technique, a controller is designed to generate sleep/gating signal when conditions are favorable for gating. A sample state diagram is shown in Fig. 3.28. A controller associated with a function unit monitors its activity on every cycle. Initially, the controller is in active state and remains here as long as the FU is busy processing inputs. If there are no inputs, then the controller switches to idle state (*idle*-1 to *idle*-N). In these states a new input to FU is handled without any delay and the state is switched back to *active*. FUs are powered on in the idle states and hence dissipate leakage power. When a sequence of idle cycles greater than a threshold N is detected, the controller goes to *sleep*-state and turns on the power gate signal to power down the execution unit saving leakage power. The unit remains shut-off until a new input requests the FU, which triggers the activation process. Since there is a certain delay (due to transient currents) in activating the FU, the controller turns off the gating signal, waits in wake-up state, and switches to active state when the FU is ready to handle inputs.

Frequent switching between power modes leads to degradation of performance as well as higher power consumption due to the transition overhead. Switching to low-power mode is considered beneficial only if the power saved in low power mode is higher than power dissipated due to transitions at acceptable performance degradation. Hence, threshold values, which represent the break-even point for power saving, are determined by considering active, idle, and sleep mode power as well as timing characteristics of the FU (for transient behavior). In general, longer sleep periods and lower idle states and wake-up time would result in higher power savings. In real applications it is observed that long idle periods are rare in case of integer units when compared to floating point units [22]. Thus, time based power

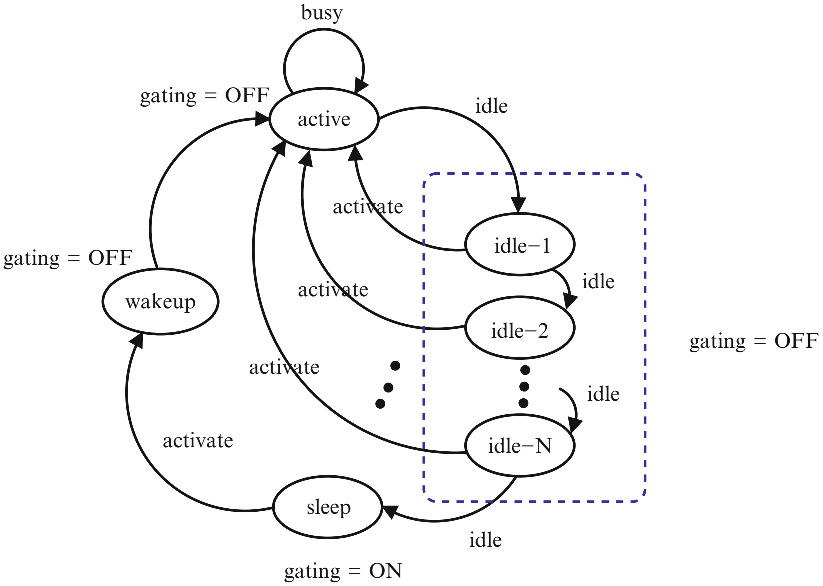


Fig. 3.28 FSM for time-based power gating. If an FU is idle for more than N clock cycles, then it is put to low-power (sleep) mode. A processing request in the sleep-state would put the FU in wake-up state for a certain time (transition delay) before switching the FU to active state

gating is not beneficial when there are no long idle periods, i.e., when the FU is used frequently, as in case of integer units.

There are many variations within time based gating, caused by defining more intermediate power modes between active and total cut-off. This technique is used to make the FU *Gradually Sleep* [13]. By having different wake-up times from each intermediate to active state and different power and performance overheads associated with state transitions, this technique presents more opportunities to save leakage power even for short idle periods.

3.5.3.2 Branch Prediction Based

In this method, time based gating is augmented with branch prediction results to determine more accurate and timely information regarding the load of execution units [22]. Aggressive speculation leads to frequent branch mispredictions, which causes the issue queue to be flushed and fetches the instructions from the correct path. During this period of time, most of the function units (both integer and floating point) are inactive. Thus, when the misprediction occurs, the FUs are immediately switched into low power mode instead of waiting for the idle period (i.e *idle-1* to *idle-N*). This allows the FUs to be in sleep mode for longer durations and thus results in more leakage power savings.

3.5.3.3 Compiler Based

The disadvantage with hardware based idle period detection is that the detection hardware itself consumes power as well as occupies area. Also, it needs some warm-up time before it can actually take the decisions on power gating. To avoid this, the detection mechanism can be offloaded to the compiler. However, this requires additional architecture support in terms of a power aware ISA. The compiler can analyze a program to determine the required ILP (and type of function units) for different code regions and insert commands/hints to power gate unnecessary function units during the execution of those code segments [39]. Thus, compiler based techniques would provide more intelligence and advance information for generating the power gating signals. This idea is explored further in Chapter 5.

3.6 Reorder Buffer

Reorder Buffer (ROB) is one of the essential structures to facilitate Out-of-Order (OoO) execution in superscalar processors. To improve instruction parallelism, superscalar processors issue instructions in an out-of-order manner. Further, due to branch speculation, some instructions from the wrong path are executed by the processor. Thus, it is very likely that instructions may complete in an order different from their actual program order. An an ROB is used to store the completed instructions temporarily, before they are committed in program-order, this also helps in supporting precise interrupts and for easy roll-back in case of mispredictions.

A simple ROB can be realized as a circular FIFO buffer. During dispatch, a tail pointer is used to allocate entries in the ROB in program order. An instruction is dispatched only when it attains a free entry in ROB. If the ROB is full then no more instructions are dispatched. After an instruction completes execution its result is stored in its allotted ROB entry. Instructions are committed from the head of the queue, pointed to by a head pointer, thus preserving the program state required for supporting precise interrupts.

In modern processors, the physical registers are integrated into ROB to support register renaming. These architectural registers are addressed either by using a rename table or by implementing an associative look-up mechanism. During instruction dispatch, operands are either read from the register file (if available), or from the ROB if the value is available but not yet committed. If it is not available in either location then a tag representing register source address is sent to the issue queue.

The ROB is implemented as a multi-ported register file, as shown in Fig. 3.29. Each ROB entry, shown in Fig. 3.29(a), consists of (a) value or result of execution, (b) instruction address (PC value), (c) address of the architecture register, and (d) a bit to specify the validity of result in the entry. Higher precision results, as in case of multiplication instruction where two 32-bit operands result in 64-bit result, would require multiple ROB entries. Thus, in such cases, twice as many ports are required

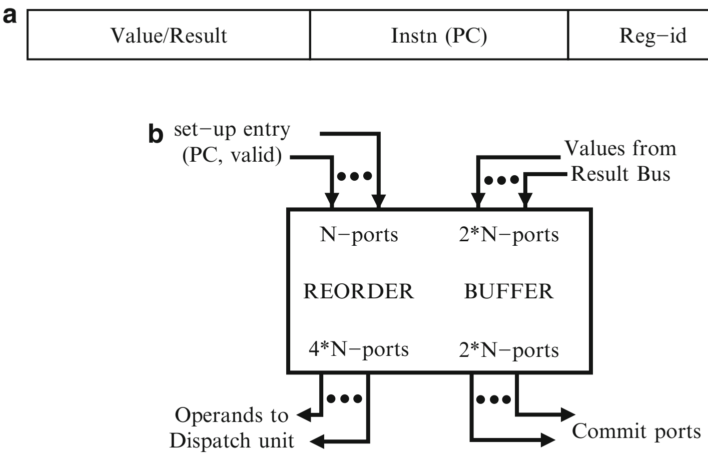


Fig. 3.29 ROB architecture. The completed instructions from FUs are written to their specific ROB entry, reserved during its dispatch. The ROB supplies the required operands to the dispatch unit. The instructions are committed in program order from the ROB's head. (**a**) Fields in ROB Entry. (**b**) Block Diagram of ROB Structure for an N-Issue Processor

to write or retrieve the value in one cycle. Register file implementation of ROB [28], shown in Fig. 3.29(b), with an N-way superscalar processor would require:

- N write ports to write set-up information such as instruction address and valid bit,
- $2N$ write ports to support a higher precision result,
- $4N$ read ports to supply operands to dispatch stage, and
- $2N$ read ports to commit the value at head of the queue.

In wide-issue processors, the port requirement for ROB along with its size makes it a complex and significantly power consuming unit in the datapath. Figure 3.30 shows the break-up of power dissipation in various components of the ROB structure [38]. Energy is dissipated in the following operations:

- *Setup:* When an instruction is dispatched, energy is consumed while locating a free entry in the ROB.
- *Dispatch:* The operand addresses are associatively looked up (or alternatively, the rename table is accessed) to check if the operand values are available in the ROB and to read the operands from ROB.
- *Writeback:* Writing the results of execution units to the ROB.
- *Commit:* Committing instructions at the head of the ROB.

The ROB being a multi-ported storage structure, many power optimizations methods that are used for register file and issue queue are also applicable for this unit. However, the criteria and conditions under which these techniques are applied to ROB are different, as discussed below.

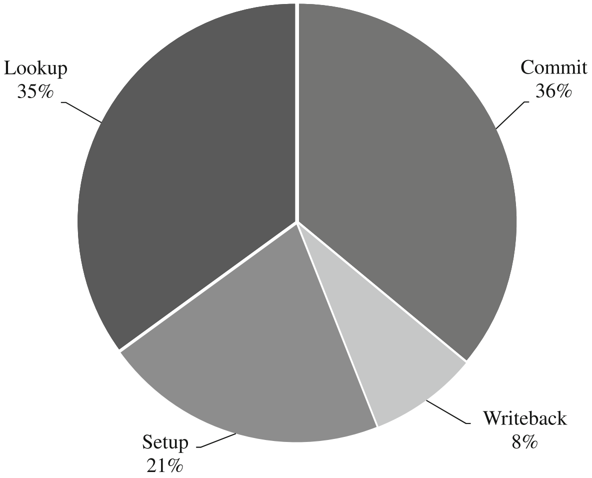


Fig. 3.30 Break-up of power dissipation in ROB

3.6.1 Port Reduction

It is observed that only a small number of operands are sourced from the ROB [28]. Most of the operands required during instruction dispatch, are derived either from the bypass paths or the register file. Simulation of various applications on different processor configurations shows that only 1-15% of the total operands required are read from ROB.

The superscalar pipeline architecture with reduced port ROB structure is shown in Fig. 3.31. In this architecture, the read ports for sourcing operands from ROB to the dispatch unit are eliminated. This architecture drastically reduces the complexity of the ROB unit and also removes the ROB access from the critical path. However by doing so, the value written to ROB is not visible to the dispatched instruction until it commits to the register file. Hence, the value is forwarded twice in this scheme, once through the result bus, when it is generated, and again when the value is committed, through forwarding buses. Instructions dispatched after the operands are generated cannot access them until they are committed to the register file. Thus, this technique degrades processor performance [28]. To reduce performance degradation, an auxiliary buffer can be used for storing a few values generated by FUs for some cycles. The auxiliary buffer is accessed by the dispatch stage for sourcing the operands. Since the number of required operands from ROB is low, the size of the auxiliary buffer could be just a few entries. Such small auxiliary structures are good enough to keep the performance degradation under acceptable limits.

With zero read ports for sourcing operands to the dispatch unit, the complexity, and hence, the power consumed by the ROB structure is reduced significantly. Through this architecture a 30% improvement in power is reported when compared to conventional ROBs [28].

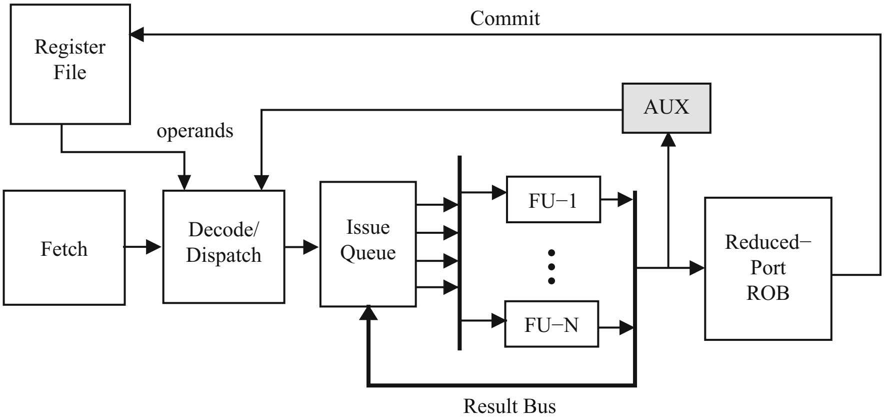


Fig. 3.31 Pipeline with reduced port ROB. A small additional buffer is used to store the result of FUs for a few cycles. This buffer is used for supplying operands to the dispatch stage. Thus, the auxiliary buffer reduces the ROB's port requirement, and hence, its power dissipation

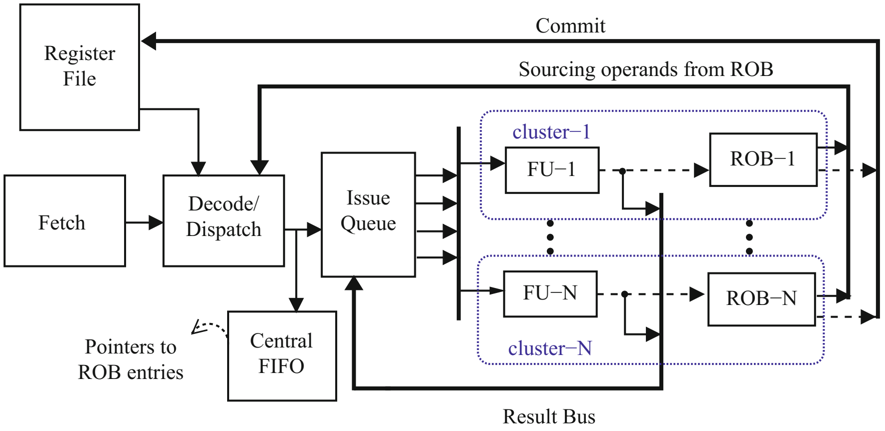


Fig. 3.32 Clustered ROB architecture. Clustering reduces the ROB's complexity, and hence, its power consumption. However, for the same result-bus bandwidth, each cluster can provide one operand per cycle which leads to a small performance overhead

3.6.2 Distributed ROB

To reduce the complexity of the ROB, the centralized structure shared by all FUs is split into multiple units. In a distributed or clustered architecture, each smaller unit of the ROB is associated with one or a group of FUs [26]. This will reduce the port requirement on the ROB structure, while improving its access time. In the clustered architecture, shown in Fig. 3.32, a centralized FIFO is still required to establish entries in the distributed ROB for the dispatched instructions. The centralized FIFO stores the association of each FU with the offset in the ROB component.

The instruction commit stage would require access through the centralized FIFO to find entries in the distributed ROB that are to be committed.

Limited bandwidth of forwarding buses allows a value to be written to the ROB component only when its associated FU acquires the bus. Each ROB component has one write port and two read ports – one for sourcing the operand to dispatch and the other for committing. Hence, when more than one operand is required from a single ROB component then the dispatch gets delayed, thereby hurting performance. However, in many real applications this situation seldom occurs and hence the IPC degradation was observed to be less than 1%.

Thus with fewer ports and simpler architecture, the distributed structure offers a power-efficient solution for the ROB unit.

3.6.3 Dynamic ROB Sizing

Exploiting the fact that program workloads vary considerably across different applications and also among various code segments of an application, many studies have proposed power optimizations by fine tuning (resizing) various datapath components. The large size of the ROB makes it a suitable candidate for dynamic sizing with varying workloads [2, 16, 37, 38].

The ROB is implemented as multiple segments, each having its own precharge logic, sense amplifiers, and input-output drivers. Each bank has independent control for activation and deactivation. With the partitioned implementation, ROB size can be dynamically varied by periodically monitoring its occupancy along with the dispatch and commit rates. As only the required ROB partition is accessed, the energy per access is reduced. Also, deactivation of unused banks leads to savings in static power. Thus, by adjusting the ROB size to the workload requirements, considerable dynamic and leakage power is saved.

3.6.4 Zero Bytes and Power Efficient Comparators

Techniques such as zero byte encoding and fast power-efficient comparators for associative search discussed in Section 3.3.2 and Section 3.3.4 respectively, are also used for reducing ROB power consumption due to similarities in the structures of the issue queue and ROB.

3.7 Branch Prediction Unit

The Branch Prediction Unit (BPU) is one of the central structures for increasing instruction level parallelism in superscalar processors. Dynamic branch prediction is is an extensively researched area, as its prediction accuracy has immense effect on overall performance of the processor. Though the branch prediction unit is not the most power hungry piece of hardware, it still consumes a notable amount (around 5-10%) of the total processor's power. This section discusses the hardware used for implementing branch prediction logic and techniques to reduce BPU power consumption.

Branches in an instruction sequence causes the processor to stall. Branch instructions are so frequent that wide issue processors may have to issue one branch instruction every cycle for achieving maximum throughput. Thus, the penalty due to branch stalls leads to significant performance loss. Dynamic hardware branch predictors are used to speculate the direction of a branch based on previous prediction history.

A simple branch predictor uses a branch history table (BHT) – a small buffer addressed by LSBs of the branch instruction address. BHT contains the information about whether the recent branch was taken or not. For an n-bit predictor, if a sequence of 2^{n-1} mispredictions is wrong then the content of the buffer is inverted. A simple 2-bit predictor results in 82% to 99% accurate predictions across many benchmarks [20].

The above scheme takes only the previous branch into consideration. Generally a branch outcome is correlated to the behavior of other branch results. Hence, to improve the prediction accuracy, the result of other branches are also used. Such predictors are called correlating predictors. An (m,n) correlation predictor uses recent m branch behaviors to choose from 2^m predictors, each of which is an n-bit predictor. For equal amount of storage, correlation predictors outperform standard predictors. The number of bits required by an (m,n) predictor is given by,

$$2^m \times n \times Number\ of\ prediction\ entries\ selected\ by\ branch\ address \tag{3.1}$$

To reduce branch penalty, an additional buffer, called Branch Target Buffer (BTB), is used for caching the target address of branches *taken*. This buffer is accessed during instruction decode stage, during which, the type of instruction is not yet known. However, if the instruction turns out to be a branch, then its target address based on speculation by the branch predictor is required before the end of decode stage. This will allow fetching the next instruction from the target location without delay, thus reducing the penalty. Figure 3.33 shows a branch target buffer that stores the instruction addresses corresponding to recent N branch instructions and their corresponding target address. A BTB entry contains the information about whether the branch is taken or not, which is used to calculate the next PC address. The PC value is sent to BTB during instruction fetch. An associative search is done in the BTB to locate an entry with the same PC address. A match indicates that the instruction is a branch and the next PC address is given by the second field (predicted PC) in the table, which corresponds to the target address of the branch. Thus, the processor starts fetching the next instruction from the address pointed to by the

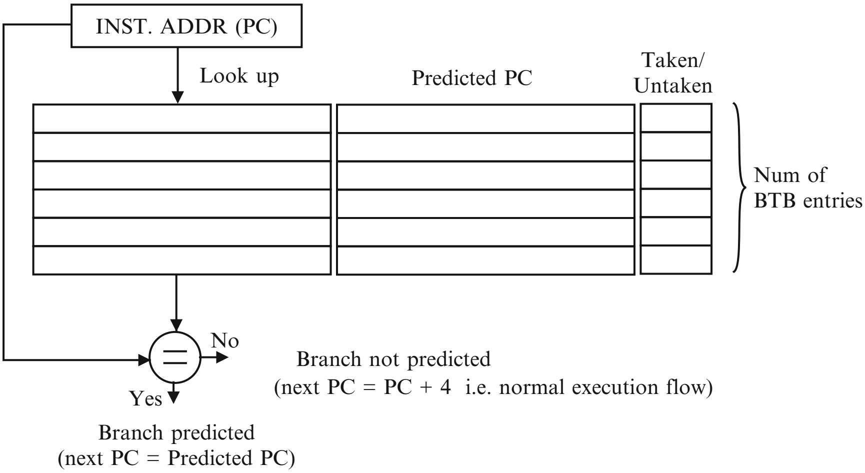


Fig. 3.33 Logical view of the branch target buffer. The instruction address (PC value) is looked up in the history table; if a match is found, then the PC is updated with the predicated value

predicted PC. The third field contains additional state bits which can be used for other prediction schemes.

Branch prediction accuracy is the most important feature of a BPU. More complex and multiple predictors along with large BTB are used to improve the accuracy. Though large BPUs have higher power consumption, they actually reduce the overall chip power by reducing wastage of power during mispredictions. For example, it is observed that a more complex BPU consumed 9% higher energy compared to simpler BPU but results in 6% net chip power savings [36]. Hence, utmost care is required while designing a low power BPU so that it does not affect the prediction accuracy. Though there could be many alternative low-power BPU architectures, only the ones that result in reduced power without degrading prediction accuracy are beneficial.

The history tables and branch target buffers could be as large as 64K-bits in size, hence consume non-trivial amount of access energy. These buffers are implemented as RAM memories for simple storage or as cache structures for associative search logic. The relatively large number of accesses into these tables/buffers causes most of the power consumption in the BPU.

Unlike other hardware units, accuracy/performance of BPU is critical for both performance as well as overall chip power. Thus, the power optimizations targeted to reduce the power consumption within the BPU have a strict restriction: the BPU accuracy shall not be compromised.

3.7.1 Banking of BHT and BTB

BTB and BHT being storage structures, power optimizations used for cache memories and SRAMs are also applicable in this case. Through banking, the central structure is split into multiple banks. Banking has several advantages:

- During access, only one bank needs to be activated [5, 36].
- Due to bit-line segmentation, the banks smaller capacitive loads resulting in lower energy per access and faster access times.
- Segmented implementation can be used to dynamically resize the memory based on application requirements [10].

All these help in reducing power consumption without affecting the BPU's accuracy. Though banking does not affect the accuracy of BPU, the price is paid in terms of increased silicon area.

3.7.2 Reducing BHT/BTB Lookups

Since lookups are performed every cycle during instruction fetch, a substantial amount of energy is expended in the look-up operation. It is observed that for many real applications, the average distance between two branch instructions is around 12 [36, 43]. Hence most of the lookups into the BHT and BTB are wasteful of energy. It is also observed that most of the cache lines do not have a control flow instruction. This is exploited to avoid unnecessary lookups into the BHT and direction prediction.

Since the BPU is accessed in parallel with the instruction cache, the additional information to avoid BHT/BTB lookups cannot be stored in the I-cache. Thus, a storage structure called Prediction Probe Detector (PPD) is added to the micro architecture, as shown in Fig. 3.34. Each entry in PPD is associated with a block in I-cache. Thus, PPD has the same number of entries as the I-cache. An entry in the

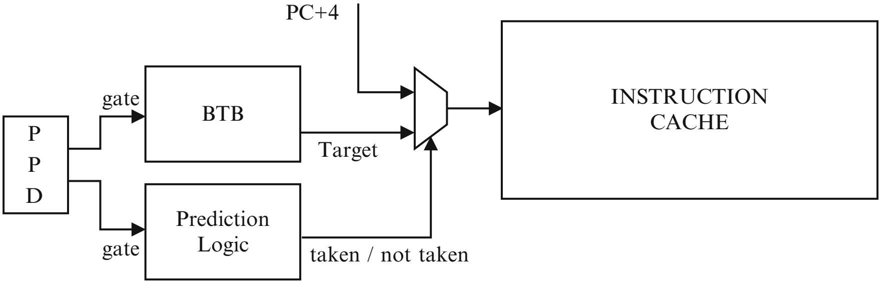


Fig. 3.34 BPU with Prediction Probe Detector [36]. Additional information regarding control instructions in cache lines is used by PPD to disable BTH and prediction logic when they are not required

PPD has two bits used for activation of BPU hardware. The PPD is updated each time the instruction cache is filled on a miss. The PPD has the information about whether the block fetched from the cache line requires a look-up and prediction or not. One bit in the PPD entry is used to gate BTB look-up operation while the other is used to gate the direction prediction logic (includes BHT). In associative caches, the line could be found in any of the allowed ways and since the PPD is accessed in parallel, it conservatively activates the BPU in cases where any of the ways have control instructions. Being small (around 4 Kb), the PPD consumes little energy as compared to the larger BPU structures and with respect to timing, these accesses could be usually accommodated into the same (fetch) cycle along with BPU access. Thus, the PPD helps in reducing the number of unnecessary and expensive lookups, thereby saving considerable (almost 30% [36]) energy in the BPU without affecting its prediction accuracy.

3.8 Summary

Continuous scaling of technology facilitated the building of denser and faster logic on the chip, which has been utilized to cater to the increasing performance demands by various applications. High performance processor architectures were designed by exploiting bit parallelism (4 to 64-bit width datapaths), instruction parallelism (scalar, vector, pipeline, out-of-order, superscalar) and thread parallelism (multi-core). With available cooling solutions, power was never a prominent constraint during the first three decades of processor evolution. Hence, the initial architectures developed were not power efficient.

The last decade saw an exponential rise in power due to increasing processor complexity and increasing leakage current. With processors dissipating more than 120 Watt power, usual cooling solutions were not suitable for handling such large heat densities. Hence, power became a major design constraint for processors. This led to drastic changes in the approach to microprocessor design. Low power methodologies were extensively studied at all design levels (device, circuit, architecture, and system). In this chapter we discussed various architectural level power optimizations used for designing high performance processors. We described in detail the break-up of power dissipated in various datapath components of a superscalar processor. In each section of this chapter we described the functionality of a datapath component and various architectural techniques used for making it power efficient. We also discussed the effect of these optimizations on the overall system performance. System level and software techniques for power reduction are covered in the next few chapters.

Apart from the architectural techniques for designing power-efficient processors, there are other important low-power techniques at device level, circuit level, system level, and also at application level. Basic low-power techniques were discussed in Chapter-2, while the system level and software techniques are covered in the next few chapters.

References

1. http://www.intel.com
2. Alper, B., Stanley, S., David, B., Pradip, B., Peter, C., David, A.: An adaptive issue queue for reduced power at high performance. In: Power-Aware Computer Systems, pp. 25–39 (2001)
3. Balasubramonian, R., Dwarkadas, S., Albonesi, D.H.: Reducing the complexity of the register file in dynamic superscalar processors. In: MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture, pp. 237–248. IEEE Computer Society, Washington, DC, USA (2001)
4. Baniasadi, A., Moshovos, A.: Instruction flow-based front-end throttling for power-aware high-performance processors. In: Low Power Electronics and Design, International Symposium on, 2001., pp. 16–21 (2001). DOI 10.1109/LPE.2001.945365
5. Baniasadi, A., Moshovos, A.: Branch predictor prediction: A power-aware branch predictor for high-performance processors. Computer Design, International Conference on **0**, 458 (2002). DOI http://doi.ieeecomputersociety.org/10.1109/ICCD.2002.1106813
6. Brooks, D., Martonosi, M.: Dynamically exploiting narrow width operands to improve processor power and performance. In: High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On, pp. 13–22 (1999). DOI 10.1109/HPCA.1999.744314
7. Brooks, D., Martonosi, M.: Value-based clock gating and operation packing: dynamic strategies for improving processor power and performance. ACM Trans. Comput. Syst. **18**(2), 89–126 (2000). DOI http://doi.acm.org/10.1145/350853.350856
8. Buyuktosunoglu, A., Y, T.K., Albonesi, D.H., Z, P.B.: Energy efficient co-adaptive instruction fetch and issue. In: In ISCA 03: Proceedings of the 30th Annual International Symposium on Computer Architecture, pp. 147–156. ACM Press (2003)
9. Chatterjee, A., Nandakumar, M., Chen, I.: An investigation of the impact of technology scaling on power wasted as short-circuit current in low voltage static cmos circuits. In: ISLPED '96: Proceedings of the 1996 international symposium on Low power electronics and design, pp. 145–150. IEEE Press, Piscataway, NJ, USA (1996)
10. Chaver, D., Pi nuel, L., Prieto, M., Tirado, F., Huang, M.C.: Branch prediction on demand: an energy-efficient solution. In: ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design, pp. 390–395. ACM, New York, NY, USA (2003). DOI http://doi.acm.org/10.1145/871506.871603
11. Correale Jr., A.: Overview of the power minimization techniques employed in the ibm powerpc 4xx embedded controllers. In: ISLPED '95: Proceedings of the 1995 international symposium on Low power design, pp. 75–80. ACM, New York, NY, USA (1995). DOI http://doi.acm.org/10.1145/224081.224095
12. Cruz, J.L., González, A., Valero, M., Topham, N.P.: Multiple-banked register file architectures. SIGARCH Comput. Archit. News **28**(2), 316–325 (2000). DOI http://doi.acm.org/10.1145/342001.339708
13. Dropsho, S., Kursun, V., Albonesi, D.H., Dwarkadas, S., Friedman, E.G.: Managing static leakage energy in microprocessor functional units. In: MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture, pp. 321–332. IEEE Computer Society Press, Los Alamitos, CA, USA (2002)
14. Farkas, K.I., Chow, P., Jouppi, N.P.: Register file design considerations in dynamically scheduled processors. In: HPCA '96: Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture, p. 40. IEEE Computer Society, Washington, DC, USA (1996)
15. Farkas, K.I., Chow, P., Jouppi, N.P., Vranesic, Z.: The multicluster architecture: reducing cycle time through partitioning. In: MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture, pp. 149–159. IEEE Computer Society, Washington, DC, USA (1997)
16. Folegnani, D., Gonzalez, A.: Energy-effective issue logic. In: Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on, pp. 230–239 (2001). DOI 10.1109/ISCA.2001.937452

17. Ghose, K., Kamble, M.: Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In: Low Power Electronics and Design, 1999. Proceedings. 1999 International Symposium on, pp. 70–75 (1999)
18. Ghose, K., Ponomarev, D., Kucuk, G., Flinders, A., Kogge, P.M.: Exploiting bit-slice inactivities for reducing energy requirements of superscalar processors. In: In Kool Chips Workshop, MICRO-33 (2000)
19. Gonzalez, R.E.: Low-power processor design. Tech. rep., Stanford University, Stanford, CA, USA (1997)
20. Hennessy, J.L., Patterson, D.A.: Computer architecture: a quantitative approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2002)
21. Hiraki, M., Bajwa, R., Kojima, H., Gorny, D., Nitta, K., Shri, A.: Stage-skip pipeline: a low power processor architecture using a decoded instruction buffer. In: Low Power Electronics and Design, 1996., International Symposium on, pp. 353–358 (1996). DOI 10.1109/LPE.1996.547538
22. Hu, Z., Buyuktosunoglu, A., Srinivasan, V., Zyuban, V., Jacobson, H., Bose, P.: Microarchitectural techniques for power gating of execution units. In: ISLPED '04: Proceedings of the 2004 international symposium on Low power electronics and design, pp. 32–37. ACM, New York, NY, USA (2004). DOI http://doi.acm.org/10.1145/1013235.1013249
23. Jouppi, N.P., Wall, D.W.: Available instruction-level parallelism for superscalar and superpipelined machines. In: ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems, pp. 272–282. ACM, New York, NY, USA (1989). DOI http://doi.acm.org/10.1145/70082.68207
24. Kim, N.S., Mudge, T.: The microarchitecture of a low power register file. In: ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design, pp. 384–389. ACM, New York, NY, USA (2003). DOI http://doi.acm.org/10.1145/871506.871602
25. Kondo, M., Nakamura, H.: A small, fast and low-power register file by bit-partitioning. In: HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, pp. 40–49. IEEE Computer Society, Washington, DC, USA (2005). DOI http://dx.doi.org/10.1109/HPCA.2005.3
26. Kucuk, G., Ergin, O., Ponomarev, D., Ghose, K.: Distributed reorder buffer schemes for low power. In: Computer Design, 2003. Proceedings. 21st International Conference on, pp. 364–370 (2003). DOI 10.1109/ICCD.2003.1240920
27. Kucuk, G., Ghose, K., Ponomarev, D.V., Kogge, P.M.: Energy-efficient instruction dispatch buffer design for superscalar processors. In: IEEE/ACM International Symposium on Low Power Electronics and Design, pp. 237–242 (2001)
28. Kucuk, G., Ponomarev, D., Ghose, K.: Low-complexity reorder buffer architecture. In: ICS '02: Proceedings of the 16th international conference on Supercomputing, pp. 57–66. ACM, New York, NY, USA (2002). DOI http://doi.acm.org/10.1145/514191.514202
29. Kursun, V., Friedman, E.G.: Low swing dual threshold voltage domino logic. In: GLSVLSI '02: Proceedings of the 12th ACM Great Lakes symposium on VLSI, pp. 47–52. ACM, New York, NY, USA (2002). DOI http://doi.acm.org/10.1145/505306.505317
30. Li, H., Bhunia, S., Chen, Y., Vijaykumar, T., Roy, K.: Deterministic clock gating for microprocessor power reduction. In: High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on, pp. 113–122 (2003). DOI 10.1109/HPCA.2003.1183529
31. Magklis, G., Scott, M.L., Semeraro, G., Albonesi, D.H., Dropsho, S.: Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In: ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture, pp. 14–27. ACM, New York, NY, USA (2003). DOI http://doi.acm.org/10.1145/859618.859621
32. Manne, S., Klauser, A., Grunwald, D.: Pipeline gating: speculation control for energy reduction. In: In Proceedings of the 25th Annual International Symposium on Computer Architecture, pp. 132–141 (1998)
33. Moore, G.: Cramming more components onto integrated circuits. Electronics Magazine **38**(8) (1965)

34. Munch, M., Wurth, B., Mehra, R., Sproch, J., Wehn, N.: Automating rt-level operand isolation to minimize power consumption in datapaths. In: Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings, pp. 624–631 (2000). DOI 10.1109/DATE.2000.840850
35. Palacharla, S., Jouppi, N.P., Smith, J.E.: Complexity-effective superscalar processors. SIGARCH Comput. Archit. News **25**(2), 206–218 (1997). DOI http://doi.acm.org/10.1145/384286.264201
36. Parikh, D., Skadron, K., Zhang, Y., Stan, M.: Power-aware branch prediction: Characterization and design. IEEE Transactions on Computers **53**, 168–186 (2004). DOI http://doi.ieeecomputersociety.org/10.1109/TC.2004.1261827
37. Ponomarev, D., Kucuk, G., Ghose, K.: Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In: Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on, pp. 90–101 (2001). DOI 10.1109/MICRO.2001.991108
38. Ponomarev, D., Kucuk, G., Ghose, K.: Energy-efficient design of the reorder buffer. In: PATMOS '02: Proceedings of the 12th International Workshop on Integrated Circuit Design. Power and Timing Modeling, Optimization and Simulation, pp. 289–299. Springer-Verlag, London, UK (2002)
39. Rele, S., Pande, S., Önder, S., Gupta, R.: Optimizing static power dissipation by functional units in superscalar processors. In: CC '02: Proceedings of the 11th International Conference on Compiler Construction, pp. 261–275. Springer-Verlag, London, UK (2002)
40. Ross, P.: Why CPU frequency stalled. Spectrum, IEEE **45**(4), 72–72 (2008). DOI 10.1109/MSPEC.2008.4476447
41. Tiwari, V., Malik, S., Ashar, P.: Guarded evaluation: pushing power management to logic synthesis/design. In: ISLPED '95: Proceedings of the 1995 international symposium on Low power design, pp. 221–226. ACM, New York, NY, USA (1995). DOI http://doi.acm.org/10.1145/224081.224120
42. Tiwari, V., Singh, D., Rajgopal, S., Mehta, G., Patel, R., Baez, F.: Reducing power in high-performance microprocessors. In: Design Automation Conference, 1998. Proceedings, pp. 732–737 (1998)
43. Yang, C., Orailoglu, A.: Power efficient branch prediction through early identification of branch addresses. In: CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems, pp. 169–178. ACM, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1176760.1176782
44. Zalamea, J., Llosa, J., Ayguadé, E., Valero, M.: Two-level hierarchical register file organization for vliw processors. In: MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture, pp. 137–146. ACM, New York, NY, USA (2000). DOI http://doi.acm.org/10.1145/360128.360143
45. Zyuban, V., Kogge, P.: Optimization of high-performance superscalar architectures for energy efficiency. In: Low Power Electronics and Design, 2000. ISLPED '00. Proceedings of the 2000 International Symposium on, pp. 84–89 (2000)

Chapter 4
Power-efficient Memory and Cache

The memory subsystem plays a dominant role in every type of modern electronic design, starting from general purpose microprocessors to customized application specific systems. Higher complexity in processors, SoCs, and applications executing on such platforms usually results from a combination of two factors: (1) larger amounts of data interacting in complex ways and (2) larger and more complex programs. Both factors have a bearing on an important class of components: memory. This is because both data and instructions need to be stored on the chip. Since every instruction results in instruction memory accesses to fetch it, and may optionally cause the data memory to be accessed, it is obvious that the memory unit must be carefully designed to accommodate and intelligently exploit the memory access patterns arising out of the very frequent accesses to instructions and data. Naturally, memory has a significant impact on most meaningful design metrics [31]:

Area Memory related structures dominate the area of most processors and ASICs. In modern processors, the cache memory structures easily account for more than 60% of the chip area.

Delay Since large amounts of program and data are accessed from memory, the access delays have an immediate impact on performance metrics such as total execution time and response time.

Power and Energy Every instruction being executed one or more memory accesses. Large amounts of data and code lead to larger energy consumption in the memory because of both the memory size and the frequency of accesses.

Power optimizations targeting the memory subsystem have received considerable attention in recent years because of the dominant role played by memory in the overall system power. The more complex the application, the greater the volume of instructions and data involved, and hence, the greater the significance of issues involving power-efficient storage and retrieval of these instructions and data. In this chapter we give a brief overview of how memory architecture and accesses affect system power dissipation, and mechanisms for reducing memory-related power through diverse means: optimizations of the traditional cache

P.R. Panda et al., *Power-efficient System Design*, DOI 10.1007/978-1-4419-6388-8_4,

memory system, architectural innovations targeting application-specific designs, compiler optimizations, and other techniques. In addition to caches used in general purpose computer based systems, we also give considerable emphasis in the chapter on power-efficient memory optimizations in ASICs and SoCs found in embedded systems.

4.1 Introduction and Memory Structure

As applications get more complex, the memory storage and retrieval plays a critical role in determining power and energy dissipation: larger memories lead to larger static power, and frequent accesses lead to larger dynamic power.

4.1.1 Overview

Figure 4.1 shows the typical external interface of a generic memory module. The interface consists of three components:

Address The address bus is an input to the memory and specifies the location of the memory that is being accessed. The width of the address bus depends on the number of memory locations. When the bus width m equals the number of address bits, it can be used to access a maximum of 2^m locations. In some memories, especially Dynamic Random Access Memory (DRAM), the address bus is time-multiplexed to carry different parts of the memory address at different times.

Data The data bus carries the associated data for the memory operation. The bus is usually bidirectional, allowing data to be either an input or output to the memory, depending on the type of operation. The width of the data bus depends on the typical datapath width used in the design. For example, if the design is

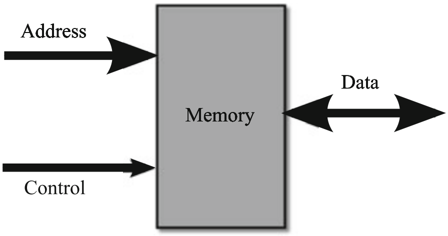


Fig. 4.1 External interface of a typical memory. The *Address* bus input gives the memory location. The *Data* bus is usually bidirectional – it is an input for a WRITE operation, and an output for a READ operation. The *Control* input consists of information such as type of operation (READ/WRITE) and chip enable

dominated by 32-bit operations, then the memory data bus is likely to be 32 bits wide. DRAM systems may be organized differently: the data bus of an individual DRAM chip usually has a smaller width (say 4 bits), and a 32-bit data bus is composed of data bits provided by a set of eight DRAM chips.

Control In addition to the address and data buses, memories usually have some control signals which are used to indicate important information such as whether the memory module is selected in the current clock cycle and an encoding of the operation to be performed. When the memory access protocol is more complex, these signals may carry other information such as clocks and row/column address strobe.

Memory accesses essentially constitute two types of operations: Read and Write.

Read The Read operation takes as input an address and returns the data stored at the corresponding memory location. The address is provided through the *address* input port in Fig. 4.1, and the data returned through the *data* port. Control signals are set to the appropriate encoding to indicate this operation.

Write The Write operation takes as input an address and the data, and stores the given data at the memory location corresponding the specified address. The address port is used as in the read operation, but the data bus is now an input port to the memory. Again, the control signals are set so as to indicate the write operation.

In addition to the above basic operations, memories may implement additional functionality such as *burst read* – fetch the data stored at a sequence of memory locations starting from a given address.

4.1.2 Memory Structure

Figure 4.2 shows a simplified view of a typical memory structure. The core storage area is organized into rows and columns of memory cells, conceptually forming a two-dimensional matrix.

The address is split into two parts as shown in Fig. 4.3: a *row address* consisting of the higher order bits, and a *column address* consisting of the lower order bits.

A Read operation can be thought of as consisting of a sequence of three phases as shown in Fig. 4.4 and Fig. 4.5. In the first phase, the row address is decoded in the *row decoder*, resulting in the activation of one of the *word lines*. This selects one row of cells in the memory, which causes transfer of data between the cell and the *bit lines* in the second phase – these lines run through all the rows of the memory. A *sense amplifier* detects the transitions on the bit lines and transmits the result on the data bus. In the third phase, a *column decoder* selects the bits from the row of cells and transmits the addressed bits to the data bus.

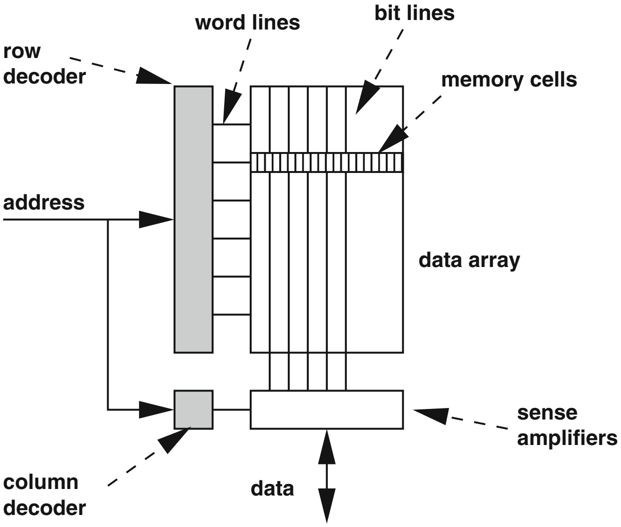


Fig. 4.2 Simplified view of typical memory structure. The most significant address bits are decoded by the *row decoder* to select a row through the *word line*. *Bit lines* attached to the selected row carry the data to *sense amplifiers*. The *column decoder* now selects the data from the right column and forwards to the data bus

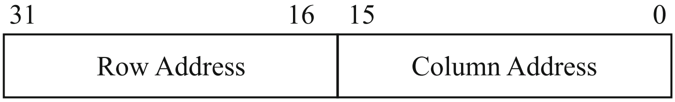


Fig. 4.3 Division of the memory address into Row Address and Column Address. The row address is the most significant part, and is used by the row decoder to select the word line. The column address is the least significant part, and is used by the column decoder to select the right data from within the selected row

4.1.3 Cache Memory

Modern applications present an inherently contradictory set of requirements from the memory system: large amounts of data have to be stored and retrieved, yet the access delay and energy dissipation should be small. Obviously, larger memories lead to longer access times and larger energy per access. To alleviate this so called *memory wall* or *memory bottleneck*, system architects usually resort to a *memory hierarchy*, consisting of several levels of memory, where higher levels comprise larger memory capacity and hence, longer access times. The memory hierarchy operates on the principle of *locality of reference*: *programs tend to reuse instruction and data they have used recently (temporal locality) and future accesses are likely to be in the same vicinity as past accesses (spatial locality)*. Thus, the first time an instruction or data is accessed, it might have to be fetched from a higher memory level, incurring

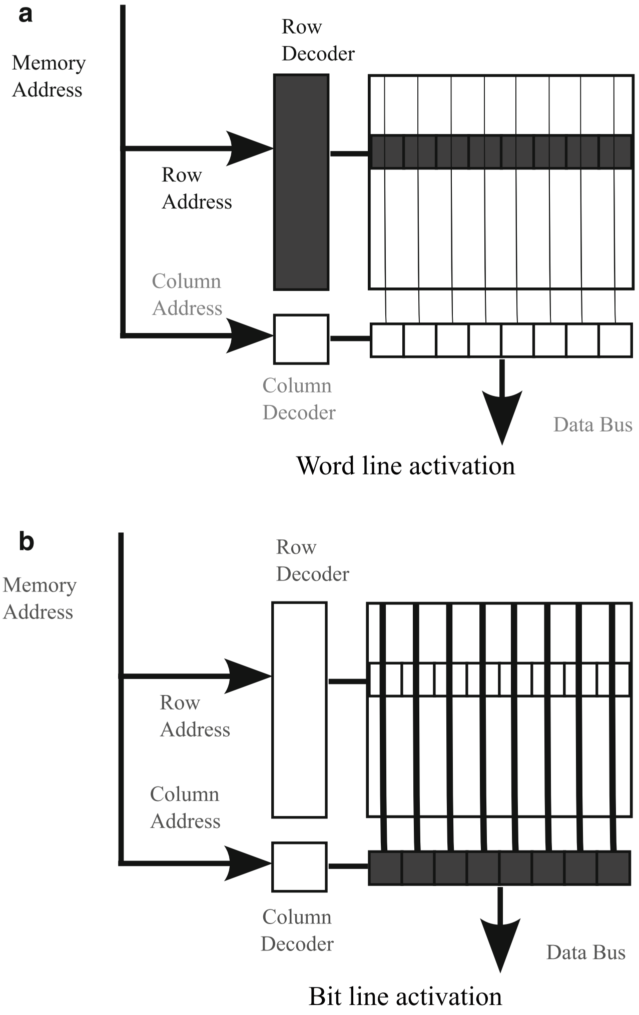


Fig. 4.4 (**a**) Phase 1 of Memory READ operation: the row address is decoded by the row decoder and a word line is activated. (**b**) Phase 2 of Memory READ: data in the memory cells are transferred via the bit lines to the sense amplifiers

a relatively higher memory access time penalty and energy dissipation. However, it can now be stored in a lower memory level, leading to faster retrieval on subsequent accesses to the same instruction or data. The different memory levels used in most processor architectures are usually: register, cache memory, main memory, and secondary memory.

Cache memory is the next memory level after registers and stores recently accessed memory locations – *instruction cache* stores recently accessed instructions and *data cache* stores recently accessed data. The two are sometimes combined into a single cache. Lower levels of cache usually reside on-chip with the processor and

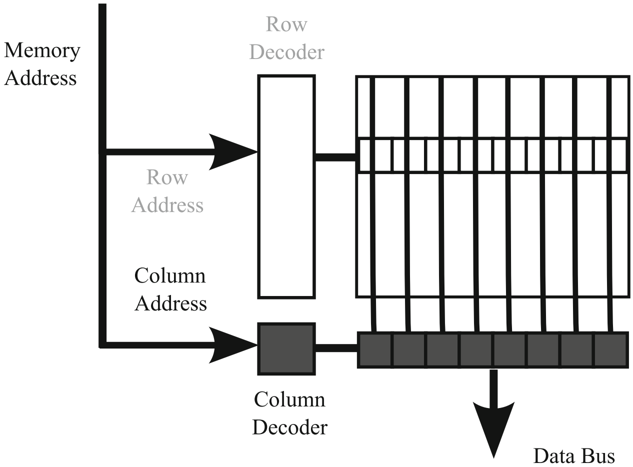


Fig. 4.5 Phase 3 of Memory READ operation: the column decoder selects the correct offset from the column address and the addressed data is forwarded to the data bus

access times for cache memory usually range from one to a few CPU cycles. On-chip caches in modern commercial general-purpose microprocessors could be as large as a megabyte. Beyond the last level of cache lies the main memory, which usually resides off-chip and is also volatile – the contents disappear when the power is reset. The main memory may be backed by some form of non-volatile secondary storage such as disk or flash memory.

Figure 4.6 shows a generalized memory hierarchy with the levels of hierarchy described above. The register file is usually incorporated into the CPU. The cache, in turn, could consist of multiple levels of hierarchy, of which the lower levels are usually located on-chip with the processor, and higher levels could be in off-chip SRAM (Static Random Access Memory). The main memory is typically implemented in DRAM (Dynamic Random Access Memory) technology, which affords higher density than SRAM, but lower access speed.

The principle of locality of reference leads to data and instructions being found in the lowest level of the cache memory hierarchy closest to the processor most of the time. When the required data or instruction is found in the level of memory that is being searched, a *Cache Hit* is said to have occurred. *Cache misses* occur when instructions or data requested by the processor are not present in the cache, and need to be fetched from the next level of the hierarchy. Cache misses can be classified into three categories [12]:

Compulsory misses. These are caused when a memory word is accessed for the first time at the current cache level. Since it is being accessed for the first time, it is obviously absent from the cache and needs to be fetched from the next level of the memory hierarchy.

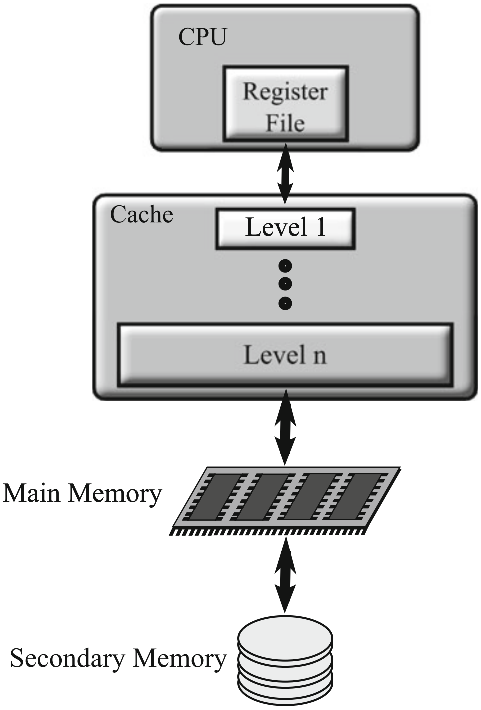


Fig. 4.6 Hierarchical memory structure: the register file is closest to the CPU, followed by the cache levels, followed by main memory (DRAM) and secondary storage (Disk, Flash memory)

Capacity misses. These are caused when cache data that would be needed in the future is displaced due to the working data set being larger than the cache. The cache designer's efforts to anticipate and store the required data from the next level may not be always successful because of the cache's limited size.

Conflict misses. These are caused when data present in the cache and useful in the future, is replaced by other data, in spite of the availability of cache space. This happens because of limitations of the mechanism used to search and replace memory words in the cache. These limitations arise out of access time constraints imposed by the system.

The *Cache Miss Ratio* is defined by the equation:

$$\text{Cache Miss Ratio} = \frac{\text{Number of Cache Misses}}{\text{Number of Cache Accesses}} \tag{4.1}$$

The cache miss ratio is a fraction between 0 and 1, often expressed as a percentage, and indicates the fraction of accesses that could not be serviced from the cache, and led to accessing of the next cache level.

Cache Hit Ratio is defined as: 1 − Cache Miss Ratio.

A *cache line* consists of a set of memory words that are transferred between the cache and main memory on a cache miss. A longer cache line reduces the compulsory misses, but increases the *cache miss penalty* (the number of CPU cycles required to fetch a cache line from main memory), and would also increase the number of conflict misses.

An elementary question that determines the working of the cache is the address mapping between the memory address and the cache location. For this purpose, the main memory is divided into blocks of the cache line size. Given a memory block address, a mapping function determines the location of the block in the cache.

The simplest cache design is a *direct-mapped* cache. Here, every memory block can be stored in exactly one cache location given by the equation:

$$\text{Cache Line} = (\textit{Block Address}) \bmod (\textit{Cache Size}) \tag{4.2}$$

where *Block Address* refers to the main memory block number and cache size is the number of lines in the cache. The mapping of memory blocks to cache locations is illustrated in Fig. 4.7, with a memory size of 1023 blocks and a cache size of 8 lines. Memory block n maps to cache line n mod 8. Since the cache is smaller, multiple blocks will map to the same cache line. Hence, the limited cache space needs to be managed effectively. In our example, suppose we access memory block 2 first, followed by block 26. Since both blocks map to the same location, block 26 displaces block 2 from the cache. Thus, if block 2 is needed later, we incur a cache miss due to the conflict between the two blocks.

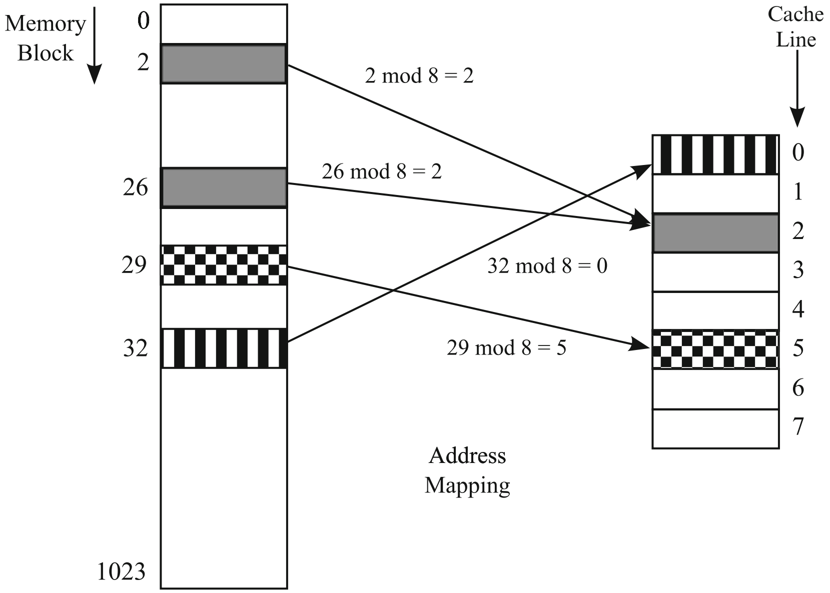


Fig. 4.7 Direct-mapped cache. For a cache with 8 lines, the cache location of memory block address n, is given by n mod 8. Cache conflicts occur when two blocks (2 and 26) map to the same cache location

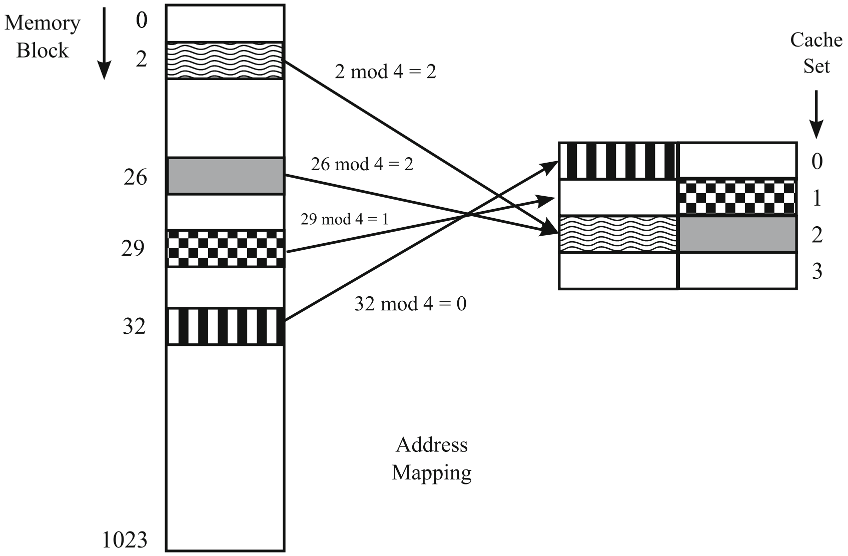


Fig. 4.8 A two-way set-associative cache. The 8 lines are divided into 4 sets of 2 lines each. The cache set for memory block address n, is given by n mod 4. The block can stay at either way of the selected set. This resolves cache conficts between the two blocks at adddress 2 and 26

Set-associative caches help reduce the cache conflict problem mentioned above. An A-way set-associative cache is divided into *sets* of A lines each. Each memory block maps to exactly one set, but within the set, the block could reside in any of the constituent A lines. The address mapping in a 2-way set-associative cache ($A = 2$) is illustrated in Fig. 4.8, with a memory size of 1023 blocks and cache size of 8 lines. Blocks 2 and 26 no longer conflict in the cache because they are accommodated in the two lines corresponding to the two cache ways.

In a fully associative cache (illustrated in Fig. 4.9), a given memory block can reside at any of the cache locations. As long as the *working set* of a program is smaller than the cache, conflict misses do not occur in these caches, but capacity misses may still occur when the *working set* of a program is larger than the cache. A fully associative cache is essentially an N-way set-associative cache (where N is the number of cache lines), whereas a direct mapped cache is a 1-way set-associative cache.

The number of cache lines in a direct mapped cache, and the number of sets in a set-associative cache, is an exact power of two (of the form 2^m), so that the mapping function is very simple to implement – the k lower order bits of the block address gives the cache line/set location.

While set-associative caches typically incur a lower miss ratio than direct-mapped ones by eliminating some cache conflicts, they are also more complex, and are characterized by longer access times, because, now, A searches need to be performed to determine if a data element exists in the cache, as opposed to a single search in the case of direct-mapped caches. Further, the additional work leads to an

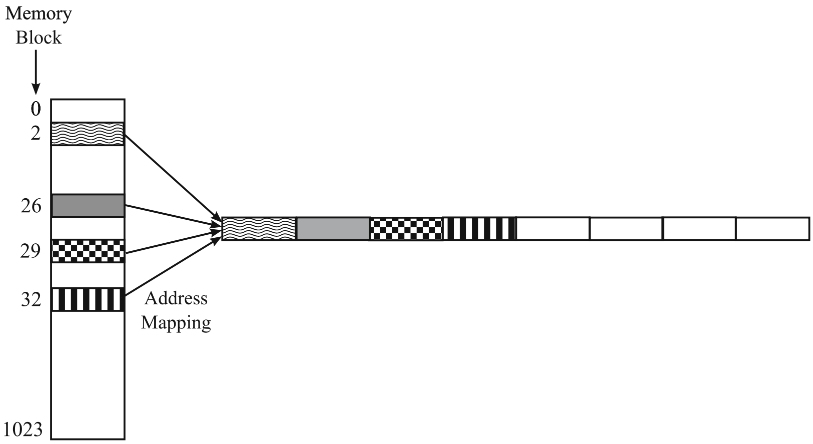


Fig. 4.9 A fully associative cache. Any memory block can reside at any of the 8 cache lines. There are no cache conflicts, but capacity misses can occur

increase in the cache energy dissipation. Conflict misses can be avoided by using a fully associative cache, but due to access time and power constraints, most cache memories employ a limited-associativity architecture.

An additional feature in associative caches is the need to implement a *replacement policy* for deciding which cache line to evict from a cache set when a new cache line is fetched. In Fig. 4.8, if another block at memory address 34 (which also maps to set 34 mod 8 = 2) were accessed, the replacement policy would help decide which of block 2 and 26 is replaced. A common replacement policy is *Least Recently Used* (LRU) [12], in which the cache line that has not been accessed for the longest duration is replaced, in keeping with the principle of locality.

4.1.4 Cache Architecture

A simplified diagram of the architecture of a typical direct-mapped cache is shown in Fig. 4.10. The memory address presented to the cache consists of three logical fields:

Offset This field, consisting of the lower order address bits, indicates which word within a cache line is to be accessed. If the cache line has 2^l words, then the offset field has l bits.

Index This field indicates the address of the set within the cache where the line will reside, if it is present in the cache.

Tag This corresponds to the higher order bits of the address, and is stored along with the data lines. The tag bits are used to identify which specific line (out of the several lines in memory which could possibly be mapped into the indexed location) currently resides in the cache.

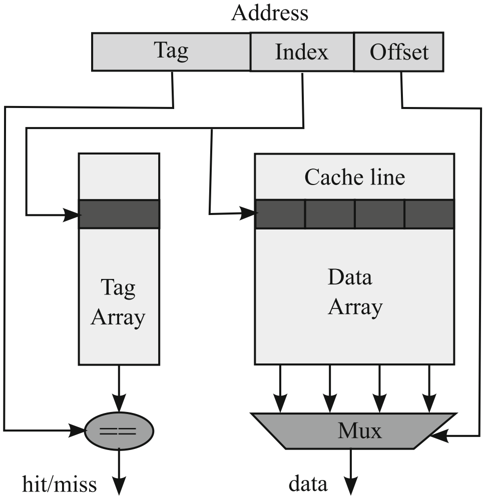


Fig. 4.10 Simplified architecture of Direct Mapped Cache. Two separate memories store the tag and data arrays. The index is used to fetch the tag and cache line from the two arrays. If the fetched tag matches the tag part of the address, then we have a cache hit, and the offset part of the address is used to select the right data from the cache line. If the tag does not match, we have a cache miss

The two major components of a direct-mapped cache are the Data Array and the Tag Array (Fig. 4.10). Suppose we access memory line L located at memory address A, and the index, tag, and offset fields of address A are given by $i(A)$, $tag(A)$, and $o(A)$ respectively. The contents of L are stored in the data array of the cache in anticipation of temporal and spatial locality, at address $i(A)$. The $i(A)$ location of the tag array contains $tag(A)$. When a new address B is presented to the cache, the cache line data at the $i(B)$ address is fetched from the data array. Simultaneously, the tag bits stored at address $i(B)$ in the tag array are also fetched. A comparator compares these stored tag bits with $tag(B)$. If the comparison succeeds, then we have a Cache Hit and the data bits read from the data array are the correct data. The offset field $o(B)$ is used to select the appropriate data from among the different words in the cache line. If the comparison fails, we have a Cache Miss and the address now needs to be sent to the next level of the memory hierarchy.

Figure 4.10 omits some other components of the cache such as control bits (Valid, Dirty, etc.) that are stored along with the tag bits in the cache. The Valid bit is used to distinguish cache contents from random values that might be stored at initialization. The Dirty bit is used to ensure that when a cache line is replaced, it is written back to the next memory level only if it is modified at the current level.

The architecture of a 4-way set-associative cache is illustrated in Fig. 4.11. There are four banks of data and tag arrays. The indexed cache line is read out from all the four data arrays. The tag bits are also read out from the four tag arrays and compared to the tag bits of the address. If there is a match with any of the stored tags, the output of the corresponding comparator will be ‘1’ (and that of the others will be ‘0’). This leads to a cache hit and the comparator output bits are used to select the data from the correct data array. If all comparisons fail, then we have a cache miss.

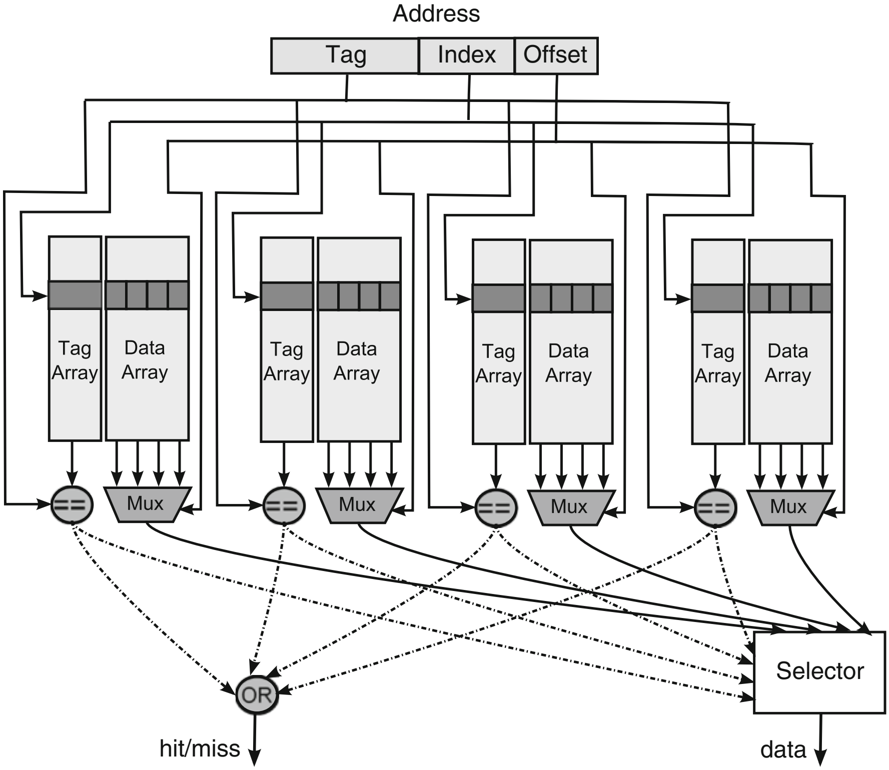


Fig. 4.11 Simplified architecture of 4-way associative cache. We have four different tag and data arrays. The index is used to fetch the tag and data from all four arrays. If the address tag matches any of the fetched tags, we have a cache hit, and the data at the corresponding cache line is selected

4.1.5 Power Dissipation During Memory Access

Power dissipation during memory accesses can be attributed to three main components:

1. address decoders and word lines
2. data array, sense amplifiers, and the bit lines
3. the data and address buses leading to the memory.

All three components are significant as each involves the driving of high capacitance wires that requires a considerable amount of energy: word lines, bit lines, and data/address buses. Power optimizations for the memory subsystem indirectly target one of these components and can be classified into the following broad categories:

- **Power-efficient memory architectures** – novel architectural concepts that aid power reduction, both in traditional cache memory design and in other unconventional memory architectures such as scratch pad memory and banked memory.

- **Compiler optimizations targeting memory power** – where code is generated for general-purpose processors targeting power reduction. This topic is explored in more detail in Chapter 5.
- **Application specific memory customization** – where the memory system can be tailored for the particular application, leading to superior solutions than a standard memory hierarchy.
- **Transformations: compression and encoding** – these are known techniques from other domains that are also applicable to memory power reduction.

How is memory optimization for high performance different from memory optimization for low power? There are some classes of optimizations that result in improving both performance and power – these are the optimizations that attempt to reduce the number of memory accesses. If the number of accesses to memory is reduced, then performance is improved because this results in reduced total latency. Similarly, reduced memory access count also means reduced energy. Most optimizations belonging to the classes of techniques summarized above specifically target low power, and are orthogonal to standard performance improving memory optimizations targeted by standard compilers. We must point out that most advanced cache architecture features that aim at improving performance by effectively reducing the number of cache misses (i.e., reducing the number of accesses to the next level of memory), also improve power as an obvious consequence. However, since they were proposed as primarily performance enhancement techniques, we do not discuss them in detail in this book. The reader is referred to a standard computer architecture text such as [12] for a more comprehensive discussion of all performance-oriented cache features that also improve power by the simple consequence of reducing the miss rate.

4.2 Power-efficient Memory Architectures

The memory subsystem in embedded processor based systems usually consists of cache memory, along with other memory modules possibly customized for the application. Because of the dominating role of instruction and data caches, new low power memory architectures have been in the area of improving traditional cache designs to make them power-efficient using a variety of techniques.

4.2.1 Partitioned Memory and Caches

Partitioning the memory structure into smaller memories (or banks) is one of the mechanisms used to reduce the effective length of the bit-lines driven during memory operations, thereby reducing power dissipation. In the multi-bank memory example shown in Fig. 4.12, the memory is physically partitioned into four banks, each with 1/4 the size of the original monolithic memory. This causes each bit line

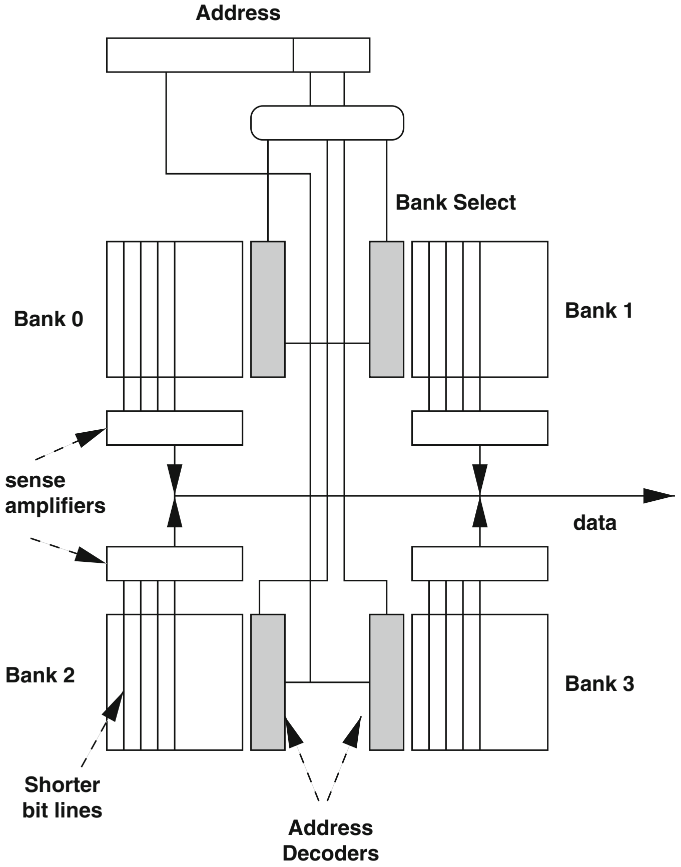


Fig. 4.12 Memory banking reduces bit-line capacitance. The memory array is physically divided into multiple banks. Each cell now needs to drive a smaller bit-line within a bank

to be of 1/4 the original length, and a proportional decrease in the switching bit line capacitance. The sense amplifiers are replicated in each bank. The lower order bits of the address (two LSB bits in Fig. 4.12) are used to generate a *select* signal to the right bank. Since only one of the banks would be activated during a memory operation, this would lead to reduced power consumption in the overall structure because of the considerably reduced energy from the switching of the smaller bit-lines.

The concept of banking is also naturally applicable to cache memory, since the bulk of the cache consists of two internal memories: the data and tag arrays. Cache banking and other partitioning studies (such as bit-line segmentation that are conceptually similar) are reported in [10, 23, 36]. One proposed variant is to make the smaller partitioned units *complete caches* as opposed to just memory banks [21]. The added flexibility here is that the different caches need not be homogeneous.

A prediction mechanism such as *most recently used* is employed to predict which sub-cache will be accessed next, and the result is used to turn the other sub-caches into low power mode.

4.2.2 Augmenting with Additional Memories

A large number of low power cache ideas have been formulated around one central principle: add an extra cache or buffer, usually small in size, and design the system to fetch data directly from this buffer in the steady state, thereby preventing an access to the L1 cache altogether. Since the buffer is relatively small, we can achieve significant power savings if we can ensure a high hit rate to the buffer.

The technique of *block buffering* [36] stores the previously accessed cache line in a buffer. If the next access is from the same line, then the buffer is directly read and there is no need to access the core of the cache. This scheme successfully reduces power when there is a significant amount of spatial locality in memory references (Fig. 4.13). The idea of a block buffer can be extended to include more than one line instead of just the last accessed line. A fully associative block buffer is illustrated in Fig. 4.14 [10]. Recent tags and data are stored in fully associative buffers associated with each set. If a new tag is found in this buffer, then the tag array read is inhibited. The matching address in the buffer is used to fetch the corresponding data from the data buffer. The correct size of this fully associative buffer will have to be determined based on an engineering trade-off because a fully associative lookup in

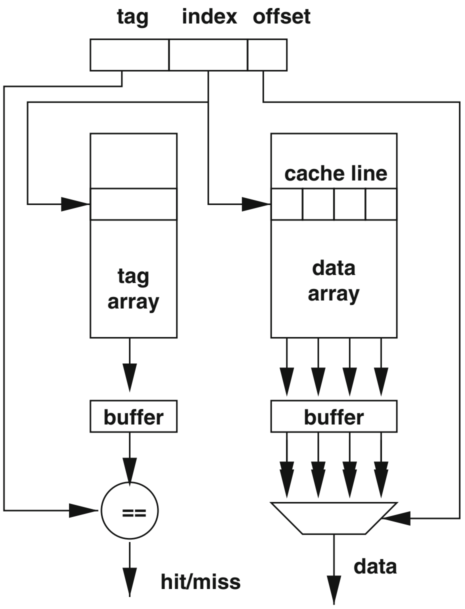


Fig. 4.13 Block Buffering. The last tag and cache line data are buffered. If the current tag matches the buffered tag, then there is no need to fetch from the memory array, thereby saving power

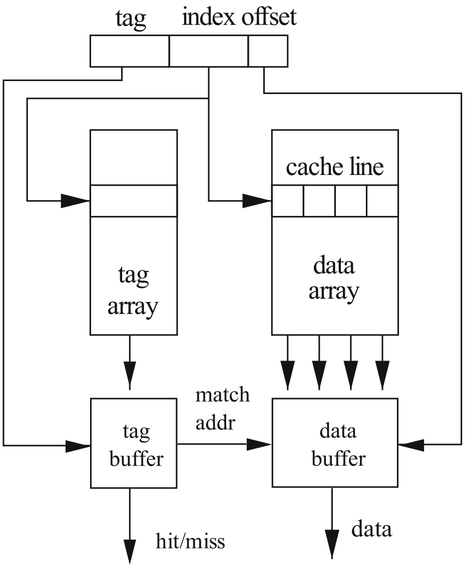


Fig. 4.14 Fully associative block buffer. An extension of the block buffer idea. Recent tags and data are stored in the buffer and looked up on a cache access. If found, then there is again no need to access the memory array. This saves power, as long as the buffer is not too big, and manages to catch a substantial number of accesses

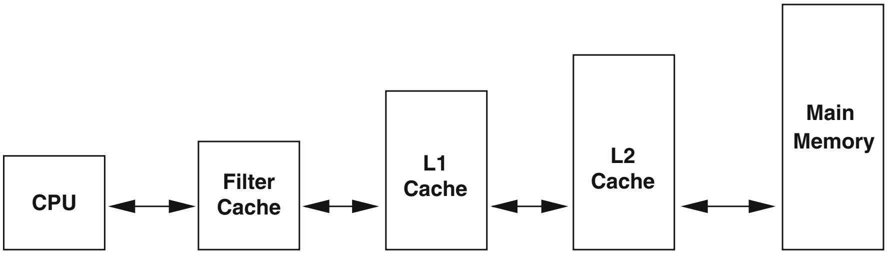


Fig. 4.15 Filter cache. A small cache placed between the CPU and the L1 cache aims at catching a significant number of accesses. Power is saved by keeping the filter cache small

a buffer is a very power-hungry operation. If the buffer is too large, then the power overheads due to the associative lookup may overwhelm the power savings due to the hit in the buffer. If the buffer is too small, then it may result in too many misses. In practice, such fully associative lookups usually restrict the size to 8 or less.

One simple power reduction strategy in caches is to introduce another level of hierarchy before the L1 cache, placing a very small cache (called a *filter cache* [22]) between the processor and the regular cache. This causes a performance overhead because the hit ratio of this small cache is bound to be lower, but it leads to a much lower power dissipation on a cache hit. If the hit ratio is reasonably high, then there may be an overall power reduction. This is illustrated in Fig. 4.15. No overall modification is proposed to the overall cache hierarchy, except that the filter cache is unusually small compared to a regular L1 cache.

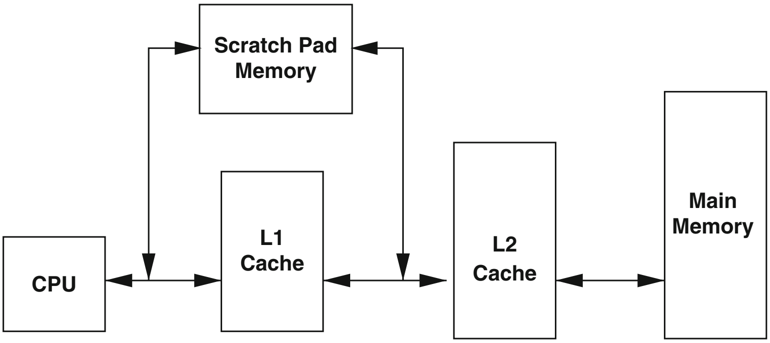


Fig. 4.16 Scratch pad memory in a memory hierarchy. The SPM is small and fast, and resides at the same level as the L1 cache. Power is saved because there is no hardware management of SPM contents, unlike in caches

Data and instructions can also be statically assigned to an additional on-chip memory whose address space is disjoint from the cached part. In *Scratch Pad Memory* [32], data is statically assigned by the compiler keeping in mind the data size and frequency of access (Fig. 4.16). Unlike in caches, scratch pad memory contents are never automatically evicted – the compiler or programmer explicitly manages the space and “hits” are guaranteed. Techniques to exploit this architectural enhancement are discussed in Section 4.4.

The well known observation that programs tend to spend a lot of time inside relatively small loop structures, can be exploited with specialized hardware. *Loop cache* [3] is one such structure consisting of an augmentation to the normal cache hierarchy. Frequently executed basic blocks within loops are stored in the loop cache. The processor first accesses the loop cache for an instruction; if it is present, there is no need to access the normal cache hierarchy, else the instruction cache is accessed. The *Decoded Instruction Buffer* [1] is analogous to the loop cache idea, but here, the decoded instructions occurring in a loop are stored in the buffer, to prevent the power overhead associated with instruction decoding. The decoded instructions are written to the buffer in the first loop iteration; in subsequent iterations, they are read off the buffer instead of the L1 instruction cache (Fig. 3.16).

4.2.3 Reducing Tag and Data Array Fetches

For performance-related reasons, the tag array and the data array in the cache are accessed *simultaneously* so that by the time the tag bits of the address of the resident cache line are fetched from the tag array and compared with the tag bits of the required address to detect hit or miss, the corresponding cache line data is already available for forwarding to the processor [12]. Thus, the fetching of the cache line data is initiated even before we know whether the access is a hit or a miss; on a miss, it is simply discarded. Since we expect most accesses to be hits, this parallel access strategy improves performance significantly. In a set-associative cache, all the tag arrays and data arrays are accessed at once. While designed for optimal performance, this overall strategy results in waste of power, since in a k-way associative cache, at least $k - 1$ fetches from the data array are discarded. Since cache lines are usually wide (cache lines of length 8-32 words are common), the power wasted here is substantial, leading to a significant scope for trade-offs between performance and power.

The simplest power optimization addressing the above issue is to sequentialize the accesses to the tag and data arrays – that is, to fetch from the data array only if the tag fetch indicates a cache hit. This prevents dynamic power dissipation incurred when data is fetched from the data array in spite of a cache miss [11]. Moreover, data only needs to be fetched from the way that matched, not from the other ways. The idea is illustrated in Fig. 4.17. Shaded blocks indicate data and tag arrays that are active in the respective cycles. In the conventional cache of Fig. 4.17(a), all tag and data arrays are shaded, indicating that all are accessed in the same cycle. In the low

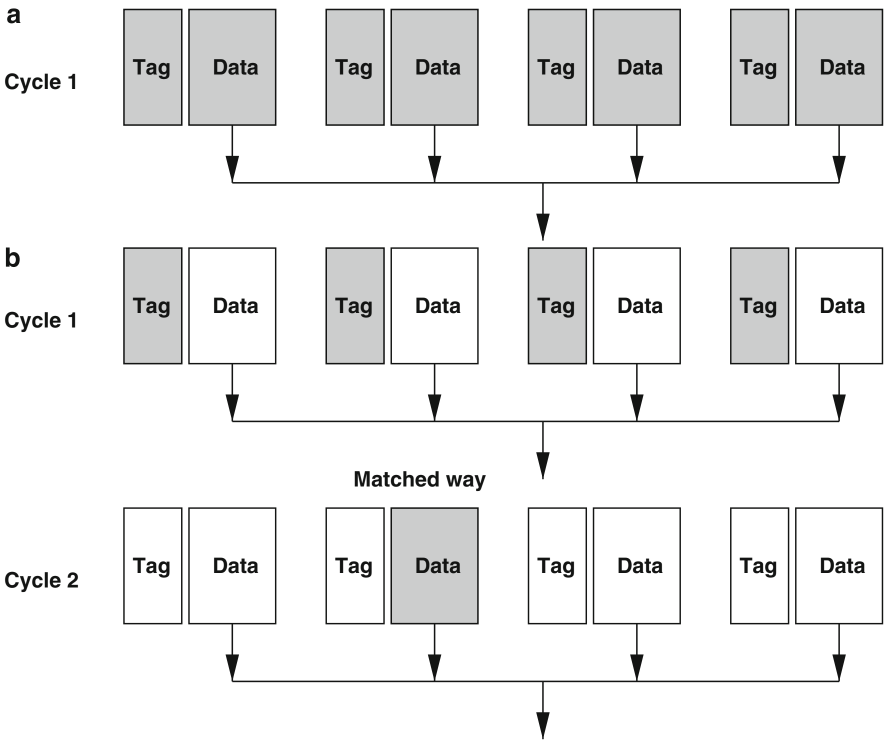


Fig. 4.17 Power saving by accessing the data array on successful tag match. (**a**) In a conventional set-associative cache, all tags and data arrays are accessed simultaneously. (**b**) When we sequentialize tag and data accesses, we first fetch only the tags. If a match is found, then data is fetched only from the matching way. Power is saved due to the avoided accesses, at the expense of time

power cache of Fig. 4.17(b), in the first cycle, all the tag arrays are accessed using the index field of the address and the tag bits are read and compared with the tag field of the address – only the tag arrays are shaded in Cycle 1 of Fig. 4.17(a). In Cycle 2, the data array of only the matched way is accessed. This leads to a performance penalty of an extra cycle, but leads to a straightforward dynamic energy reduction due to the three ways for which the data arrays are not accessed.

The above approach reduces cache access energy but compromises on performance. Another simple idea is (in case of instruction cache) to retain the address of the last accessed cache line, and to fetch from the tag array only if the next instruction refers to a different line [33]. If the reference is to the same line, then we are guaranteed a cache hit, and power is saved by preventing the redundant access of the tag array. The above is similar to the block buffering strategy, but can be generalized in an interesting way: we can assert that if there has been no cache miss since the last reference to a basic block of instructions, then there is no need to fetch the tag from the instruction cache in the next reference (since the previously fetched instruction has not had an opportunity to get evicted yet). The information about whether the target of a branch instruction exists in the cache is recorded in the Branch Target Buffer, a commonly used structure in modern processors. If the condition is satisfied, then the fetch from the tag array is disabled, saving memory activity in the process [15].

The observation that, in set-associative caches, consecutive references tend to access data from the same way, can be exploited in a mechanism that predicts the way number for the next access to be the same as the previous one. On the next access, only the tag and data arrays of the predicted way are activated, resulting in a significant amount of dynamic power savings when the prediction is correct [14]. When the prediction turns out to be incorrect, the rest of the ways are fetched in the next cycle, which incurs a performance penalty. This is illustrated in Fig. 4.18.

An alternative method of incorporating way prediction is through the *location cache* – a proposal for the L2 cache [28]. This is an extra cache that is used to indicate which way in the actual cache contains the data. A hit in the location cache indicates the way number, and hence, we only need to access the specific way in the L2 cache, thereby avoiding reading all the tag and data arrays of the set-associative cache. A miss in the location cache indicates that we do not have a prediction, and leads to a regular access from all the ways in the L2 cache. This is illustrated in Fig. 4.19. The location cache needs to be small in order to ensure that its power overheads do not overwhelm the saved power.

A certain amount of flexibility can be built into set associative caches to control accesses to the different ways – ways can be selectively enabled or disabled depending on the characteristics of an application. For example, in the L2 cache, we can reserve separate ways for instruction and data so as to prevent conflicts. Also, for small programs where instruction cache conflicts are not expected, some of the ways assigned to instructions can be disabled to reduce activity in their tag and data arrays [26]. In the *way-halting cache* [38], some least significant tag bits from each way are stored in a separate array, which is first accessed and the corresponding bits

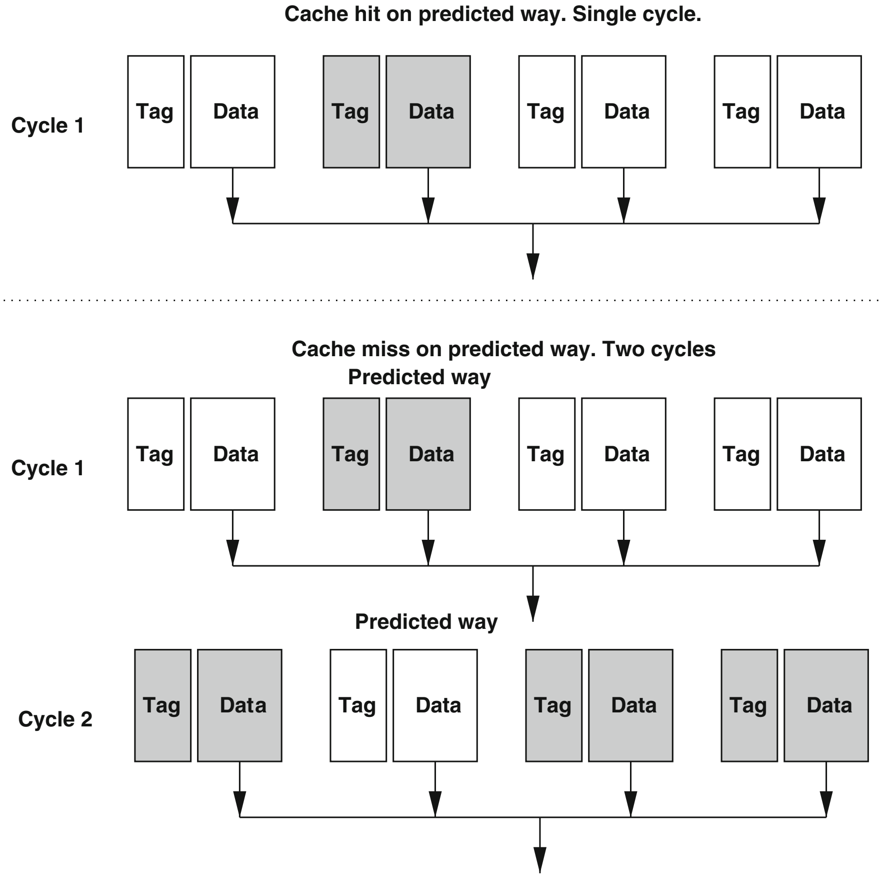


Fig. 4.18 Power saving by Way Prediction in associative caches. Tag and data are fetched only from the predicted way. If prediction is correct, this reduces power by avoiding accesses to all the other ways. If incorrect, then all other ways are accessed, losing some time

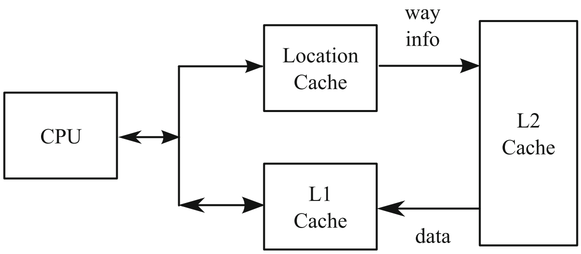


Fig. 4.19 A Location Cache stores the predicted L2 way. When L2 is accessed, only the predicted way is looked up first

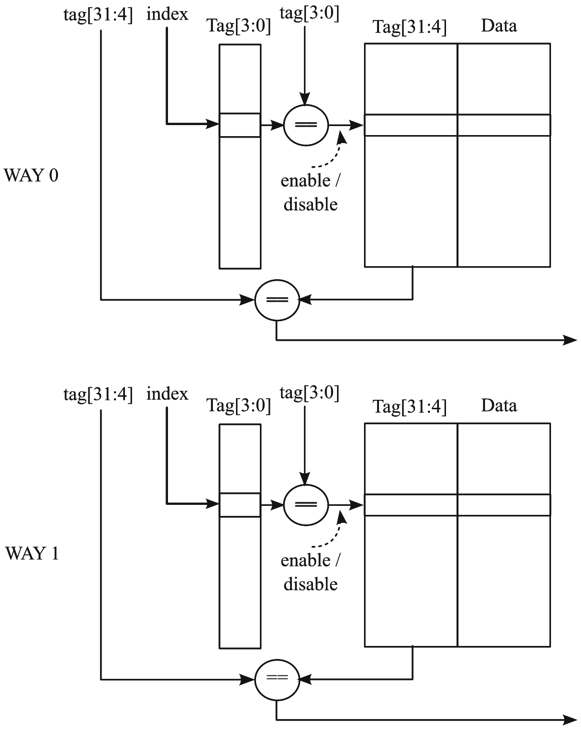


Fig. 4.20 Way-halting cache. A subset of the tag bits are first compared for equality. If unequal, then there is no need to fetch the remaining tag bits

of the address compared. If the comparison fails, then a matching failure is guaranteed for that way, and hence, the actual fetch of the tag and data arrays is prevented, saving power. This is shown in Fig. 4.20. The *enable/disable* signal is generated for each way by comparing the least significant 4 bits of the tag. On mismatch, the fetching of the remaining tag bits and data are disabled.

4.2.4 Reducing Cache Leakage Power

The techniques discussed in previous sections target dynamic power consumption in caches. As mentioned earlier, the importance of static power has been growing in recent years, and with smaller geometries, the contribution of static power to the overall power consumption is growing. Static power is dissipated as long as a voltage is supplied to the circuit, and can be eliminated by turning power supply off (in which case memory data is lost) or reduced by turning the voltage down (in which case data can be retained, but accessing the data requires us to raise the voltage again). A few strategies have been proposed to address the static power dissipation in caches.

An important observation regarding lifetime of cache contents is that, the data tends to be accessed frequently in small intervals of time, and is essentially dead for large periods when it is never accessed. This leads to an interesting question – can we turn off power to a cache line if we predict that the line will not be accessed in the near future? The *cache decay* technique [20] relies on this approach. A counter is maintained for each line; when it reaches a threshold value with no access to the cache line, the power to the line is turned off after updating the next level cache with the new value if necessary. The counter is reset on an access to the cache line. To keep the overhead of maintaining the counters low, there is only one global counter, but a two-bit derived counter is placed in each line to control the power supply. The threshold value of the counter is determined from the values of the static energy dissipated and the energy expended in re-fetching the line from the L2 cache. This is illustrated in Fig. 4.21.

An alternative technique to turning off the power to cache lines is to turn down the voltage so that data is retained, but cannot be accessed directly. In order to access the data, the line would have to be first switched to high voltage (causing a performance overhead). The power saved per line in this manner is smaller than that in the decay technique where the power supply to the line is turned off, but this may permit more lines to be moved into the *drowsy* state [8]. The idea, called *drowsy cache*, is proposed to be used for all lines in the L2 cache, and some lines of the L1 cache. A simple strategy of putting the entire cache to drowsy mode once in a while works quite well, as opposed to introducing active circuitry to maintain counters on a per line basis. A variation on this theme is to use predictive schemes to selectively move lines to and from drowsy state. Information about cache lines with high temporal locality is maintained in the BTB, and these lines are kept active. When a cache line is accessed, the sequentially following line is moved to active state, anticipating its use. A very fine grain control can be exercised by permitting the compiler to insert instructions to turn individual cache lines off when its use is not anticipated for a long time [13, 39].

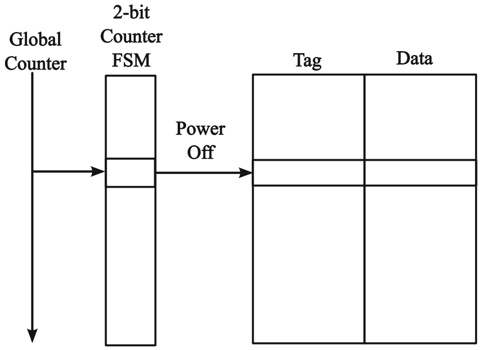


Fig. 4.21 Cache decay. If a cache line has not been accessed for some time, then turn off the power to the line

4.3 Translation Look-aside Buffer (TLB)

The concept of virtual memory was developed to relieve the programmer of the burden of managing physical memory during program execution. The operating system automatically manages the loading and relocation of the program at execution time.

For efficient memory management, the address space is divided into *pages* which are analogous to cache blocks/lines, but at a higher level of granularity. The page size typically varies from 4096 to 65536 bytes. A CPU with virtual memory generates a virtual address which gets translated into a physical address using hardware and/or software approaches. This is referred as *address translation*. A *page table* is used for mapping virtual address to physical address as shown in Fig. 4.22. The page table contains mapping information of virtual pages to the available physical pages. Thus, the page table size depends on the available physical memory size and the size of each page. Typical page tables could be as large as 4 MB and are hence usually stored in main memory. A miss in virtual memory would require an access to secondary memory – usually hard disk – which may exhibit access latencies of millions of processor cycles. Since the miss penalties are very high, a fully-associative strategy is used for placing blocks in the main memory.

Since the translation would impose an additional memory access overhead on every memory access, fast address translation is performed using a special on-chip cache called *Translation Look-aside Buffer* (TLB). TLB is a small cache that stores the virtual to physical mapping information of the most recently accessed memory locations. Typical TLB sizes vary from 8 to 128 entries. The structure of a TLB is similar to that of a normal cache, with the tag array containing the virtual address that is looked-up, and the data array containing the physical address to which it maps. When the CPU requests a memory access, the virtual address is looked up

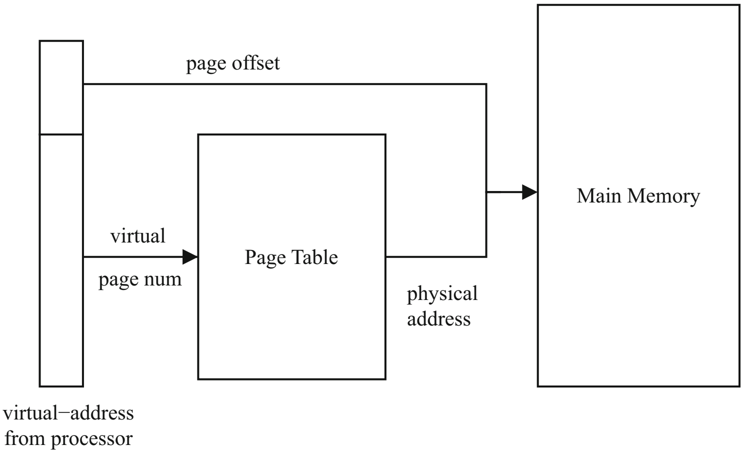


Fig. 4.22 Address Translation: the virtual address generated by the CPU is translated into a physical address using the page table

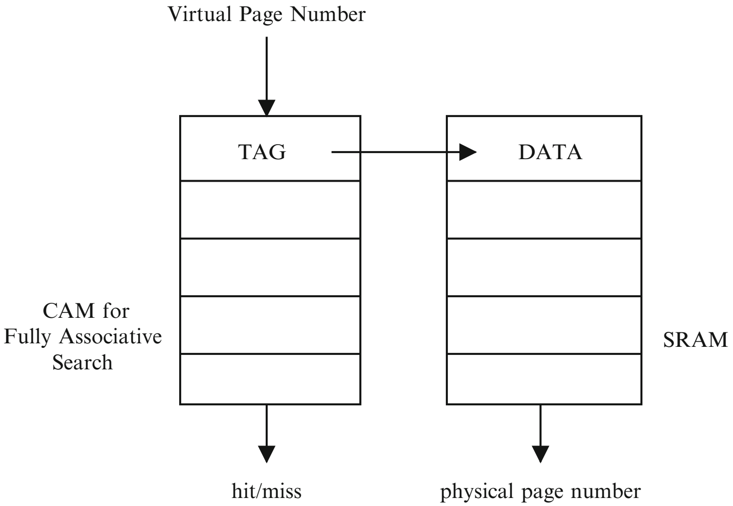


Fig. 4.23 TLB architecture: logical view of the TLB structure

simultaneously in both the L1-cache as well as the TLB. In order to simplify the hardware, TLBs for instruction and data memories are kept separate.

The instruction TLB (ITLB) is used in every cycle. Since consecutive instructions are usually mapped to the same page, the translation lookups for instruction references can be optimized. In case of data TLB (DTLB), the number of lookups per cycle depend on the number of parallel data accesses supported. For a wide issue superscalar processor, the TLB would require multiple read ports. The logical structure of the TLB is shown in Fig. 4.23. To support fully associative lookup, a content addressable memory (CAM – generally considered to be expensive in terms of power) is used for storing the tags (virtual addresses). A simple SRAM is used for storing data (the corresponding physical address). In a well managed memory system, TLB miss rate is very small – usually less than 1%.

The basic cell implemented for CAM logic would require 9 transistors (shown in Fig. 4.24), compared to 6 transistors for a simple SRAM memory cell. It consists of the standard 6 transistor (6T) structure for storage and three additional transistors for comparing the stored bit with the content on bit lines. All bits in the tag share a common *match* signal. During CAM lookup, all the match lines are precharged and on a mismatch they are discharged by the NOR transistor at the bottom.

In a TLB lookup operation, a maximum of one entry would result in a match, which is then used for reading the corresponding data from the SRAM. Hence, during a lookup, every tag entry (except the matched one) would charge and discharge the match lines leading to a large power dissipation. Larger cell size and matching lines lead to a CAM cell occupying larger area and consuming higher power than a standard SRAM cell. The fully associative structure leads to significant power consumption in the TLB, in spite of it having relatively small number of entries in comparison to caches. In embedded processors with small cache sizes, TLB power

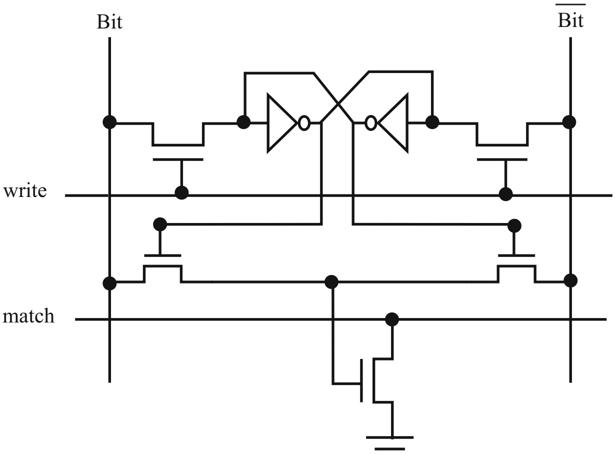


Fig. 4.24 CAM cell: in the 9-transistor structure of a CAM cell, the standard 6T structure is used for storing the bit while the additional transistors are used for comparing the content with the bit lines for a match

forms one of the major components of the total system power consumption. For instance, in the StrongARM processor, TLB is reported to account upto 17% of the total power.

In this section we discuss some of the important optimizations and trade-offs considered in designing power-efficient TLBs.

4.3.1 TLB Associativity – A Power-performance Trade-off

Ideally, we would like the TLB to be fully associative to minimize the accesses to main memory and hence obtain both power and performance benefits. However, it is well established that the returns in terms of cache hits actually diminish with increasing cache sizes. Since the TLB is also inherently a cache, this principle is also applicable to TLBs. As a consequence, power dissipated in the fully associative search for every lookup in a large cache can overshadow the power saved from reducing the accesses to the main memory. Hence, large fully associative TLBs can have a negative impact on power with little performance improvement. The trade-off between power and performance needs to be carefully evaluated while designing a TLB for a system.

4.3.2 Banking

TLB banking is an attractive low power solution [17, 27]. In a banked architecture shown in Fig. 4.25, the TLB is split into multiple tag and data banks. Each tag

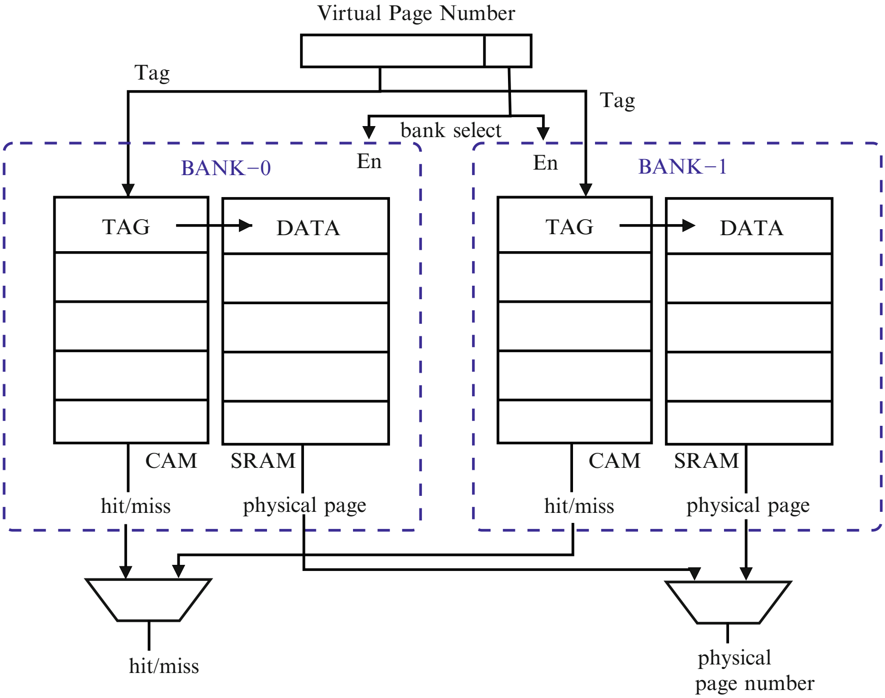


Fig. 4.25 2-way Banked TLB architecture: In a banked TLB structure, some bits of the virtual address are used for bank selection. Each bank has a fully associative tag storage and the corresponding data entries in SRAM. During lookup only one bank is activated, thereby reducing the access energy

bank is fully associative, while each data bank is a traditional SRAM. A lookup request to TLB would need to access only one bank, thereby reducing the access energy per lookup by N times, where N is the number of banks. For the same TLB *size* (number of entries), an N-way banked architecture would have an associativity of $\frac{size}{N}$ which would result in a performance loss. Thus, unlike traditional cache architectures, banked TLB architecture needs careful study of power-performance trade-offs.

Some advanced architectural enhancements and allocation schemes for reducing performance loss due to banking are discussed below:

Victim Buffers – A victim buffer can be used to hold N recently replaced TLB entries, similar to the concept used in caches. The victim buffer is shared by all banks. During a TLB lookup operation, the victim buffer and the appropriate TLB bank are searched in parallel. A hit either in the bank or victim would result in a TLB hit. Thus, victim buffers can alleviate most of the capacity misses caused due to banking and hence improve the performance [5]. The size of the victim buffer should be selected in a way that the power consumed by searching the buffer is small compared to the power saved due to reduced main memory accesses.

Allocation Policy – A more aggressive allocation policy can be used while replacing TLB entries in a banked architecture. In general, if a free entry is not available in the selected bank to hold the mapping for a new page, an existing page is replaced. In a more aggressive policy, the replaced entry could be placed in other banks instead of being discarded, even at the expense of a second replacement. During a TLB lookup operation, first the TLB bank corresponding to the address being looked-up is searched; on a miss, all the other banks are searched. If the entry is not found in any of the banks, then a request to the main memory is sent. This scheme would essentially ensure the same miss rate as that of a fully associative search. However, it is more power efficient if the "hit ratio" to the bank that is searched first is reasonably high, which is generally the case [5].

4.3.3 Reducing TLB Lookups

Since each TLB lookup is very costly in terms of power, intelligent techniques that reduce number of lookups without having an adverse impact on performance form attractive power optimization alternatives.

4.3.3.1 Deferred Address Translation

By employing virtually indexed and virtually tagged caches at the L1 level, address translation would be required only during an L2 access (assuming L2 is physically addressed). Thus, TLB needs to be accessed only on L1 misses. Though this would require an extra cycle for all L1 cache misses, the number of accesses, and hence, energy dissipation in the TLB is considerably reduced [18]. Similarly, if the L2 cache is also virtually indexed and virtually tagged, then the translation could be further deferred and could even be implemented in software by the operating system.

4.3.3.2 Using Address Mapping Register

Modern processors employ separate TLB structures for instructions and data to allow concurrent lookups for both data and instruction references. The ITLB is used whenever an instruction reference requires address translation. Due to the temporal locality property of instruction addresses, there is a very high probability that successive accesses would belong to the same memory page. With page sizes of 4KB to 64KB, one can expect a large number of accesses to the same page before proceeding to the next. This property is exploited by storing the mapping for the most recently accessed page in a special hardware register. During an ITLB lookup, this register is accessed first for address translation and only if it is not found in this register, the power hungry TLB lookup operation is performed [18]. Since this is stored in a register close to the processor, the overhead in timing on a miss is negligible. The concept is similar to the idea of block buffering (Section 4.2.2).

4.4 Scratch Pad Memory

Scratch Pad Memory (SPM) refers to on-chip memory whose contents are explicitly managed by the compiler or programmer [32]. A typical architecture is shown in Fig. 4.26. Address and data buses from and to the CPU could lead to on-chip caches and scratch pad memory, but both these memory modules are optional. If the data or instruction requested by the CPU is present on-chip, in either the scratch pad or the caches, then it is accessed from the respective module. Otherwise, the next level of memory hierarchy (off-chip) is looked up. The implementation of the scratch pad could be in either SRAM or embedded DRAM on chip. The main logical characteristic of the scratch memory is that, unlike caches where the management of the memory content is decided transparently by hardware, in scratch pad the management is explicitly performed by the compiler or programmer. This could have both positive and negative consequences. The advantages are that data and instructions stored in the scratch pad are guaranteed to be present where they were last stored, until they are explicitly moved, which makes access times more deterministic. This not only enables predictability that is of crucial importance in real-time systems, but also simplifies the hardware considerably – there is no need for the tag memory storage, access, and lookup, which saves energy per access when data is

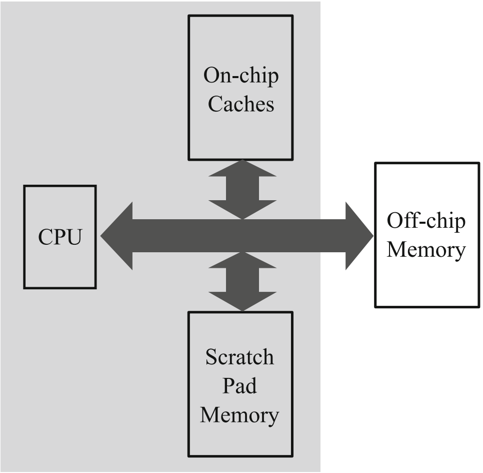


Fig. 4.26 Scratch pad memory. The CPU's request for on-chip data/instruction can be served from either SPM or on-chip cache. Management of SPM contents must be performed in software

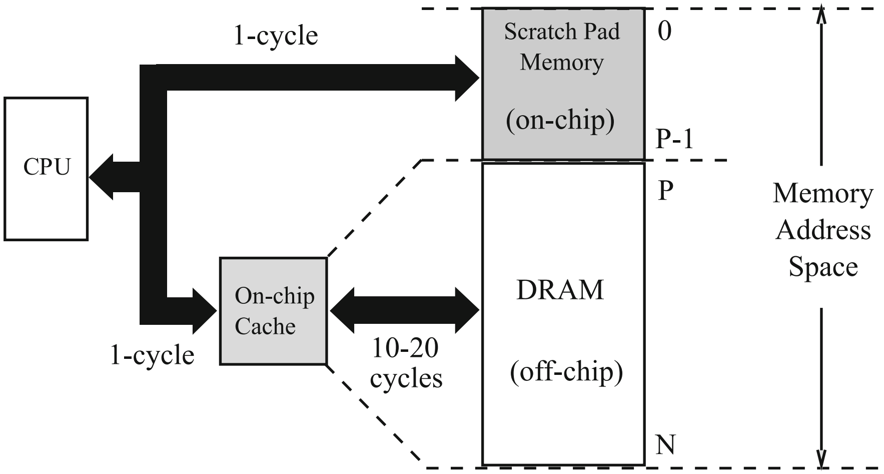


Fig. 4.27 Address mapping in scratch pad memory. Addresses $0..P-1$ are in SPM. Addresses $P..N-1$ are accessed through the cache. Access to both SPM and on-chip cache can be assumed to be fast

found in the scratch pad. The latter makes scratch pads more energy efficient than caches as long as the memory contents are efficiently managed statically. The disadvantages of SPM based architectures is that, often, program and data behavior may not be easily analyzable by the compiler, making it difficult for it to exploit the SPM well.

A given architectural platform could omit either the scratch pad or the caches. In this section we will assume both are present on-chip, but most of the decision making process about data and instruction mapping into scratch pad memory remains unchanged even if the on-chip caches are absent. Figure 4.27 shows a typical view of the memory address space $0..N-1$ divided into on-chip scratch pad memory and off-chip memory (typically implemented in DRAM). Addresses $0..P-1$ are mapped into scratch pad memory, and $P...N-1$ are mapped to the off-chip DRAM and accessed through the cache. The caches and scratch pad are both on-chip and result in fast access (1 cycle in Fig. 4.27). Accesses to the DRAM, which occur on cache misses, take relatively longer time and larger power (access time is 20 cycles in Fig. 4.27). If the cache is not present in the architecture, then the connection to off-chip DRAM is usually through a Direct Memory Access (DMA) engine.

4.4.1 Data Placement in SPM

A significant role can be played by the compiler when the architecture contains scratch pad memory structures, as these memories are directly managed by the compiler. Compile-time analysis is involved in determining which data and instructions should be mapped to the scratch pad memory. Since SPM space is limited, we would like to store relatively critical data in it [32]. Criticality can be defined in terms of two major factors:

- size of data – smaller data sizes are preferred for SPM.
- frequency of access – higher frequency is preferred for SPM.

A problem of this nature maps approximately to the well known *Knapsack Problem* in computer science [9], where we are given a knapsack of a fixed size, and can fill it with objects of different sizes and profit values. The objective is to select a subset of the objects that fit into the knapsack while maximizing the profit. In the SPM application of the knapsack problem, the SPM size is the equivalent of the knapsack size; object size corresponds to the data/array size; and the profit can be measured in terms of the number of accesses to the variables mapped into the scratch pad. This is because SPM access involves less energy dissipation than a cache access; this is a consequence of the SPM being a simple SRAM with no additional tag-related circuitry characterizing the cache, and hence, no associated dynamic power dissipation. In terms of performance, the guaranteed “hit” to the scratch pad ensures no cache-miss related delays.

Standard knapsack heuristics can also be applied in a straightforward manner to the SPM data allocation problem. The *profit density* metric, defined as P_i/S_i characterizes each object in the knapsack problem in terms of the profit per unit size. The greedy heuristic fills the knapsack by objects in decreasing order of profit density, as long as the object size is smaller than the remaining knapsack space. The same approximate heuristic can also be used for SPM allocation, where we sort all arrays in terms of access frequency per unit array size, and consider arrays for SPM assignment in decreasing order. Scalar variables can be all stored in the SPM, as they may not amount to too much total space.

The above simple formulation can be used to obtain a reasonable SPM allocation of arrays, but several other factors can also be taken into account in a more comprehensive SPM allocation solution. First, several arrays can reuse the same SPM space because their *lifetimes* can be non-overlapping. Secondly, when there is a possibility of conflicts in the caches between different arrays accessed repeatedly, one of them could be diverted into the SPM to ensure good overall memory access behavior for both arrays. An example is shown in Fig. 4.28. Here, arrays *a* and *b* are accessed in a regular manner, whereas accesses to *c* are data-dependent. Cache conflicts between arrays *a* and *b* could be avoided by suitably aligning the start positions of the arrays

```
for (i = 0; i < 100; i++) {
   b [i] = a [i] + 1; // regular access to 'a', 'b'
   c [b [i] ] = a [i]; // irregular access to 'c'
}
```

Fig. 4.28 Arrays with irregular accesses could benefit from SPM allocation. Arrays *a* and *b* are accessed regularly, and if properly laid out, should exhibit good cache behavior. However, access to *c* is irregular – not much locality might exist, and *c* could benefit from SPM allocation

in memory. However, unpredictable and unavoidable cache conflicts with array *c* could occur. The conflicts could be avoided by assigning *c* to the SPM, where it is guaranteed to not interfere with the cache contents.

4.4.2 Dynamic Management of SPM

We observe that the SPM allocation strategies in Section 4.4.1 assign an array to the SPM for the entire duration of its lifetime. This has some obvious disadvantages. An array may be occupying the SPM even if it is currently not being used, thereby precluding a different, more relevant, array from occupying the valuable space.

Figure 4.29(a) shows an example with arrays *a* and *b* accessed in two different loops, with *a* being allocated to the SPM, and *b* not allocated. This causes the second

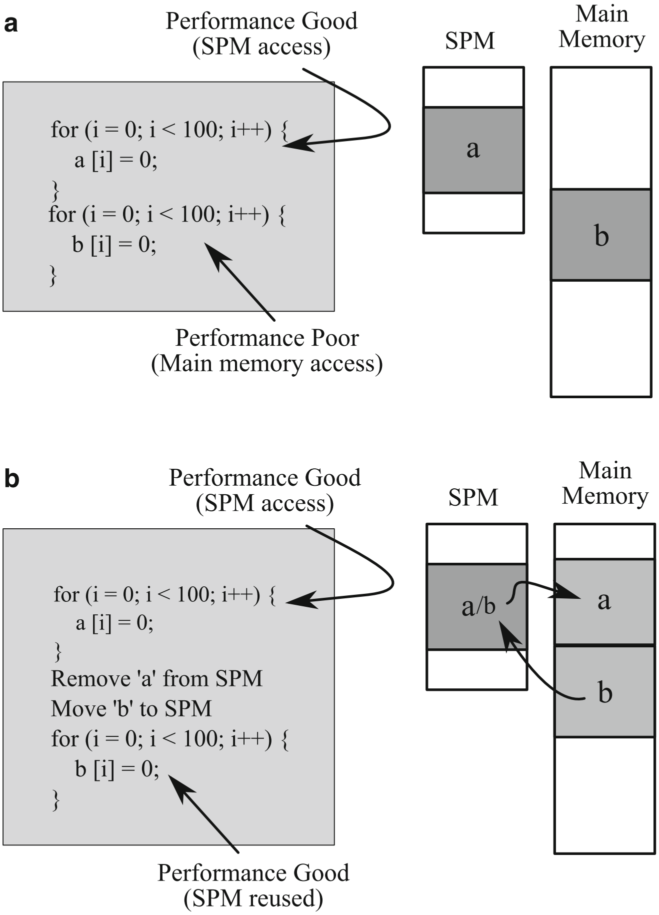
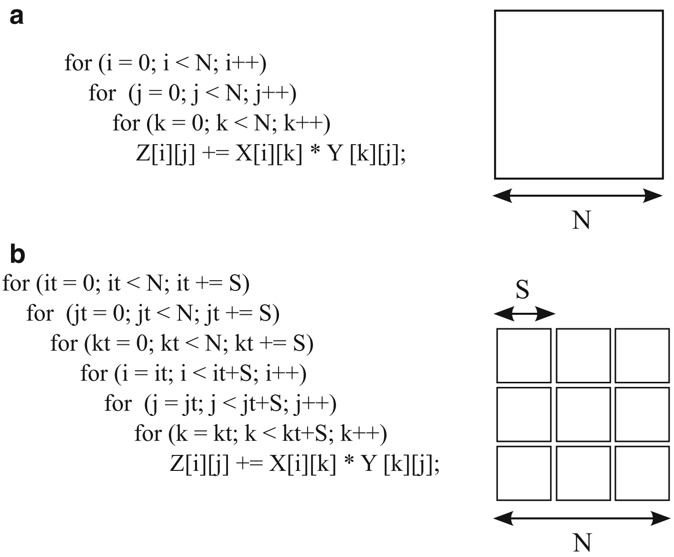

a
Performance Good (SPM access)
SPM
Main Memory
for (i = 0; i < 100; i++) {
a [i] = 0;
}
for (i = 0; i < 100; i++) {
b [i] = 0;
}
a
b
Performance Poor (Main memory access)
b
Performance Good (SPM access)
SPM
Main Memory
for (i = 0; i < 100; i++) {
a [i] = 0;
}
Remove 'a' from SPM
Move 'b' to SPM
for (i = 0; i < 100; i++) {
b [i] = 0;
}
a/b
a
b
Performance Good (SPM reused)

Fig. 4.29 Space reuse in scratch pad memory. (**a**) Without reuse of SPM space, only one of *a*, *b* fits into SPM. The second loop performs poorly. (**b**) After the first loop, *a* is replaced in the SPM by *b*

loop to perform poorly. However, noticing that *a* is not accessed in the second loop, we could substitute *a* by *b* before control enters the second loop (Fig. 4.29(b)). This calls for a more general strategy for identifying program points where we transfer data between the SPM and the background memory. The decision of whether to transfer an array to the SPM would depend on a comparison of the expected performance and energy improvement from fetching the data against the overhead incurred in doing so [37].

The next relaxation called for in the dynamic management of SPM is to permit a portion of an array to occupy space in the SPM. This allows us to assign SPM space to arrays that have heavy reuse but are larger than the SPM. One useful entity in this context is the idea of a *block* or *tile* – a portion of a multi-dimensional matrix that appears in the context of the *blocking* or *tiling* compiler transformation. Computations involving large matrices often show poor performance when the arrays are too large to fit into the data cache. Spatial and temporal locality exist in the computation, but capacity misses prevent data reuse. To overcome the problem, the arrays are divided into small blocks or tiles that fit into the cache, and the loops are appropriately restructured. This results in a significant performance improvement. Loop tiling is illustrated in Fig. 4.30 with a matrix multiplication example.

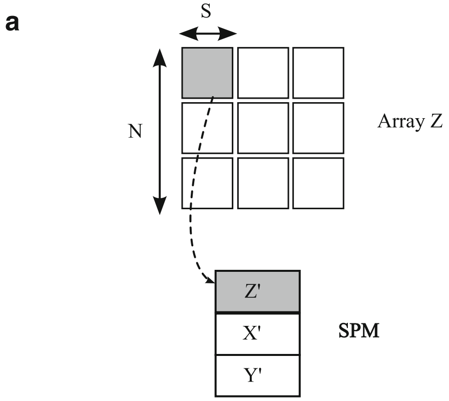
A similar conceptual transformation can be effected in the SPM. Arrays can be divided into tiles, moved into the SPM before being processed, and moved back later. The process is illustrated in Fig. 4.31 with the same matrix multiplication example. Array tiles are first transferred into the SPM with the READ_TILE routine. After processing, the tile Z' is written back to memory with the WRITE_TILE routine [19].



```
for (i = 0; i < N; i++)
    for  (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            Z[i][j] += X[i][k] * Y [k][j];
```

```
for (it = 0; it < N; it += S)
    for  (jt = 0; jt < N; jt += S)
        for (kt = 0; kt < N; kt += S)
            for (i = it; i < it+S; i++)
                for  (j = jt; j < jt+S; j++)
                    for (k = kt; k < kt+S; k++)
                        Z[i][j] += X[i][k] * Y [k][j];
```

Fig. 4.30 Blocking or tiling transformation. (**a**) Original loop: arrays might be too large to fit into on-chip memory. (**b**) Tiled loop: tile size is chosen so that it fits into on-chip memory



b

```
for (it = 0; it < N; it += S)
  for  (jt = 0; jt < N; jt += S)
    for (kt = 0; kt < N; kt += S) {
       READ_TILE Z[it...it+S-1, jt...jt+S-1] -> Z'[0...S-1][0...S-1]
       READ_TILE X[it...it+S-1, kt...kt+S-1] -> X'[0...S-1][0...S-1]
       READ_TILE Y[kt...kt+S-1, jt...jt+S-1] -> Y'[0...S-1][0...S-1]
       for (i = 0; i < S; i++)
         for  (j = 0; j < S; j++)
           for (k = 0; k < S; k++)
             Z'[i][j] += X'[i][k] * Y'[k][j];
       WRITE_TILE Z'[0...S][0...S] -> Z[it...it+S-1, jt...jt+S-1]
    }
```

Fig. 4.31 Tiling transformation in SPM. (**a**) Tiles are first transferred to the SPM. (**b**) Updated ones are written back to main memory after processing

4.4.3 Storing both Instructions and Data in SPM

An argument for scratch pad memory allocation can also be made in the context of instructions. Frequently executed instructions or basic blocks of instructions can be mapped to SPM so as to prevent the energy and performance-related overheads due to being evicted from the instruction cache. Power is saved both on account of the elimination of tag storage and access, and both performance and energy improves because of reduced cache misses. In fact, a unified formulation can use the same scratch pad memory to map either instructions or data [16, 35].

4.5 Memory Banking

The presence of multiple memory banks creates interesting optimization opportunities for the compiler. Traditionally a few DSP processors used a dual-bank on-chip memory architecture, but in modern systems, banking is used in various contexts for various objectives. In synchronous DRAMs (SDRAMs), banking is used to improve performance by keeping multiple data buffers from different banks ready for data access. In application specific systems, dividing a monolithic memory into several banks leads to considerable performance improvement and power savings. The performance improvement comes from the ability to simultaneously access multiple data words, while the power savings arise from smaller addressing circuitry, word lines, and bit lines, as observed earlier.

The power optimization problem is to assign data to memory banks in order to minimize certain objective functions. In terms of performance, we would like to be able to simultaneously access data in different banks so that computation time decreases, assuming multiple datapath resources are available. In terms of power dissipation, we have the possibility of moving specific banks to *sleep mode* during periods of inactivity.

Figure 4.32 illustrates the memory bank assignment problem with a simple example. In the schedule shown in Fig. 4.32(a), nodes in dark, labeled A, B, etc., represent memory load and store nodes, and result in memory accesses. $M1$ and $M2$ are the two memory banks. Initially, variables A, C, E, F, and G are assigned to bank $M1$. Variables B and D are assigned to $M2$. Assume that the memory banks can be either in *active* mode or *sleep* mode. Further, assume that the memory can transition from active to sleep state instantaneously, whereas it requires one cycle to transition from sleep to active mode (i.e., one cycle has to be spent in *wake-up* mode during this transition). The power dissipation during active and wake-up modes is high, and the power during sleep mode is low. The schedule in Fig. 4.32(a) does not permit any transition to sleep modes due to the lack of sufficiently long periods of inactivity. In order to exploit the sleep mode without compromising the schedule length, we can alter the bank assignment. Let us interchange the bank assignment of C and D, which results in an alternative schedule shown in Fig. 4.32(b). We notice that now bank $M2$ is inactive for a sufficiently long time, permitting us to move it to sleep mode for two cycles, before returning to active mode via one cycle in wake-up mode [25].

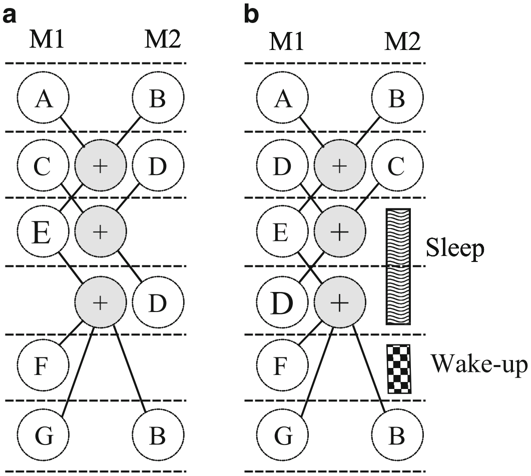


Fig. 4.32 Bank assignment targeting power efficiency. Transition of a bank from *active* to *sleep* state is instantaneous, while transition back requires one cycle. (**a**) Bank assignment affords no possibility to move either bank into sleep state. (**b**) Variables re-assigned to permit $M2$ to be moved to sleep state for two cycles

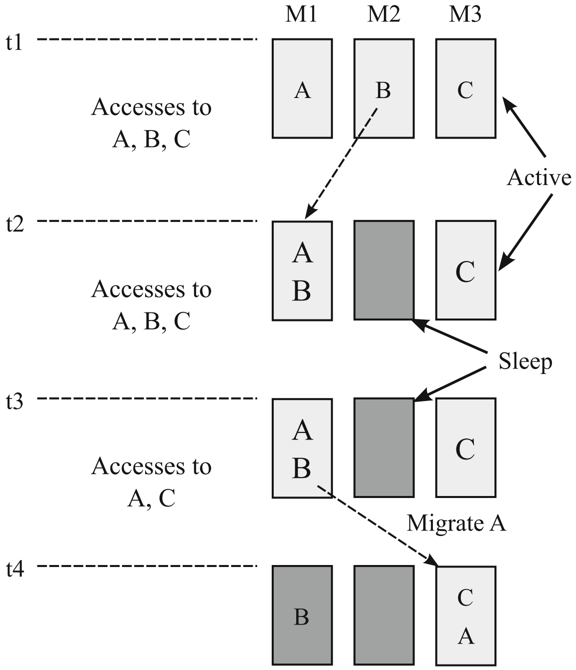


Fig. 4.33 Data migration. At $t2$, B is moved to $M1$, permitting us to move $M2$ to sleep mode. B is not accessed between $t3$ and $t4$. At $t4$, A is moved to $M3$, permitting us to move $M1$ to sleep mode

Often, the presence of conditionals prevents a proper static analysis of decisions relating to the setting of memory banks to sleep mode for saving power. Dynamic approaches involving the run-time *migration* of data to different memory banks may be needed. We can keep track of variable- and bank-referencing patterns, and can attempt to cluster recently referenced variables into the same bank, thereby creating an opportunity to maximize the number of inactive banks and move then to sleep mode.

Figure 4.33 illustrates the basic idea of a data migration strategy. Initially, at time $t1$, three arrays A, B, and C are stored in three memory banks M1, M2, and M3. Our monitoring hardware detects accesses to both A and B between times $t1$ and $t2$, enabling the migration of array B to module M1. Assume that M1 is large enough to accommodate A and B, but is not large enough to accommodate all three arrays. With this migration, module M2 can be set to sleep mode. Between $t2$ and $t3$, we detect accesses to all of A, B, and C, so we just retain the current modes. Between $t3$ and $t4$, we detect accesses to A and C. This leads us to migrate A to module M3, causing both M1 and M2 to be set to sleep mode.

In order to ensure the effectiveness of the data migration approach, some issues that need to be addressed are:

- the delay and energy overhead of the data being migrated needs to be accounted for; large array variables can result in large overheads.
- data migration needs to account for the sizes of the memory banks.

- additional hardware is needed to keep track of temporal correlation of different variables. Since this can lead to a significant power overhead if done every cycle, a sample-based approach may be necessary [6].

4.6 Memory Customization

One of the most important characteristics of embedded SoCs is that the hardware architecture can be customized for the specific application or set of applications that the system will be used on. This customization can help improve area, performance, and power efficiency of the implementation for the given application. Even when the overall platform may be fixed, different subsystems can be independently tailored to suit the requirements of the application. The memory subsystem is a fertile ground for such customization. Cache memory can be organized in many different ways by varying a large number of parameters: cache size, line size, associativity, write policy, replacement policy, etc. Since the number of possible configurations is large, an explicitly simulation-based selection strategy may be too time-consuming. A static inspection and analysis of the application can reveal several characteristics that help determine the impact of different parameter values without actual execution or simulation.

Data caches are designed to have cache lines consisting of multiple words in anticipation of spatial locality of memory accesses. How large should the cache line be? There is a trade-off in sizing the cache line. If the memory accesses are very regular and consecutive, i.e., exhibit good spatial locality, a longer cache line is desirable, since it saves power and improves performance by minimizing the number of off-chip accesses and exploits the locality by pre-fetching elements that will be needed in the immediate future. On the other hand, if the memory accesses are irregular, or have large strides, a shorter cache line is desirable, as this reduces off-chip memory traffic by not bringing unnecessary data into the cache. An estimation mechanism could be used to predict the impact of different data cache line sizes based on a compiler analysis of array access patterns. The cache line size is bounded from above by the DRAM burst size, which represents the maximum number of data words that can be brought into the cache with a single burst access.

A given amount of on-chip data memory space could be divided in various ways into data cache and scratch pad memory. An associated memory customization problem is to determine the best division of the space. The division that results in the least off-chip memory accesses would again maximize performance as well as minimize power. Figure 4.34 shows a typical variation of the total number of off-chip memory accesses with increasing data cache size (D), with the total on-chip memory fixed to a constant T. Thus, the choice of a larger cache size D results in a correspondingly smaller scratch pad memory size $T - D$. We note that when the cache size is too small or too large, the number of memory accesses is relatively higher. When the cache size is too small, it is essentially ineffective due to serious capacity misses. When the cache is large, occupying all of the on-chip memory, then there is no

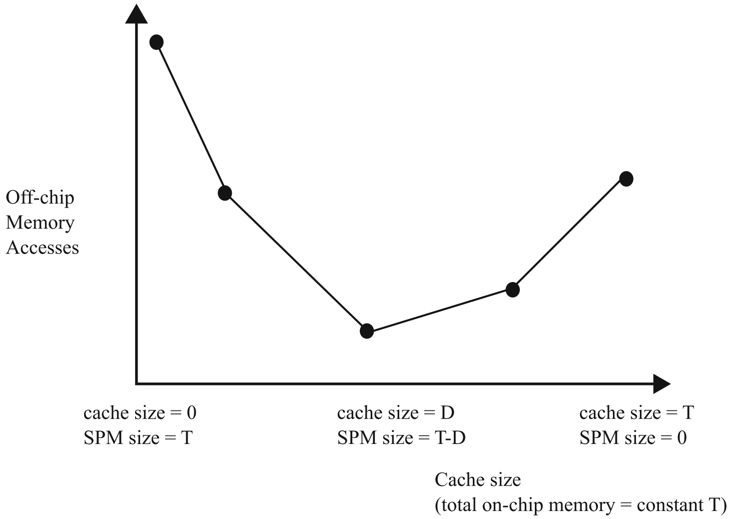


Fig. 4.34 Division of on-chip memory space T into cache and scratch pad. When cache is too small, latency increases because of capacity misses. When cache is too large, latency also increases because there is no room for effective SPM utilization

room for scratch pad memory, thereby losing the advantages of SPM. The optimal on-chip memory configuration lies somewhere in between the two extremes, with some space devoted to both data cache and SPM, augmented by an intelligent compiler strategy that allocates data to the two components.

Other than the cache vs. scratch pad trade-off above, the total on-chip memory space allocated to the application can also be a variable, to be customized depending on the requirements of the application. In general, data cache performance, as measured by hit rate, improves with increasing cache size. Similarly, increasing scratch pad memory performance also leads to higher performance. A memory exploration loop can vary the total on-chip memory space, and study the performance variation, as depicted in Fig. 4.35. Each data point (A, B, C, D, and E) could represent the best result obtained from a finer-grain exploration of scratch pad vs. cache partition for a fixed total on-chip memory size [30]. Figure 4.35 shows that the performance, as measured by hit rate (or, equivalently, in terms of total off-chip memory traffic), improves with increasing on-chip memory size, but tapers off beyond a certain point. Design points such as C in Fig. 4.35 are strong candidates, since they represent the boundary beyond which it may not be worth increasing the memory size – for higher sizes, the hit-rate improves marginally, but the resulting larger memory size represents an overhead in both memory access time and energy.

Embedded SoCs allow the possibility of customized memory architectures that are tailored to reducing power for a specific application. Figure 4.36 shows an example of such an instance. A simple loop is shown in Fig. 4.36(a), with two arrays *a* and *b* accessed as shown. A default computation and memory architecture for implementing this system is shown in Fig. 4.36(b). The computation is performed in

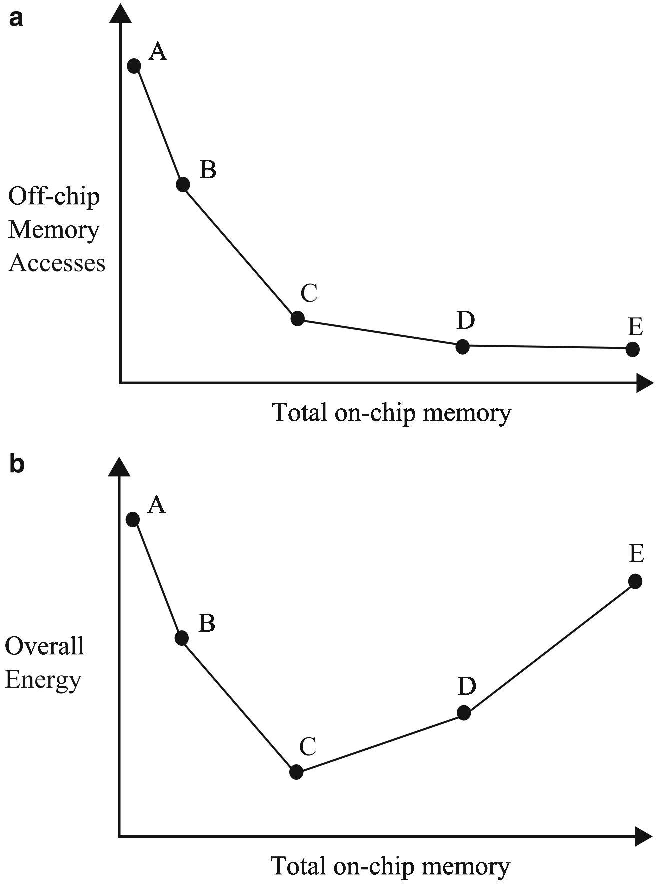
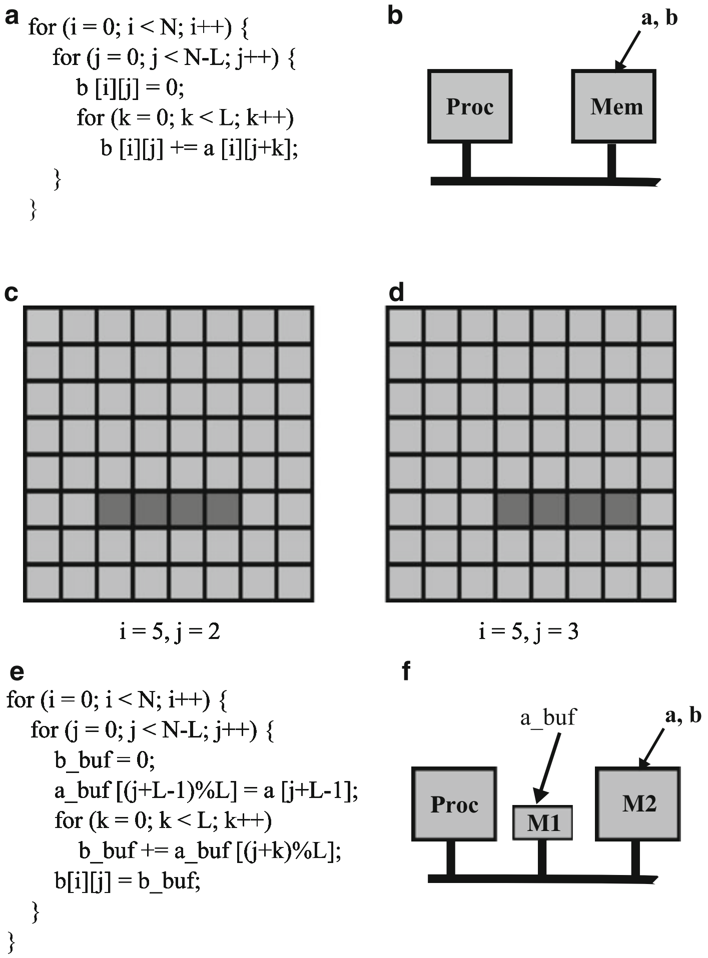


Fig. 4.35 (**a**) Off-chip memory access count decreases with increasing on-chip memory size. Beyond point D, performance does not improve (but power continues increasing). (**b**) Energy decreases with increasing on-chip memory, but the trend reverses for larger on-chip memory because the per-access energy of larger memories is greater

the *proc* block, and arrays a and b are stored in memory module *Mem* of size N words. Assuming there is no cache structure here, every array reference results in an access to *Mem* block, resulting in the standard power dissipation associated with reading or writing of an N-word memory.

A more careful analysis of the array reference patterns reveals some optimization possibilities [29]. Figure 4.36(c) shows the elements of a that are accessed in one iteration of the j-loop, with $i = 5$ and $j = 2$. We assume that a is an 8×8 array and $L = 4$. In Fig. 4.36(d), we show the elements of a accessed in the next j-iteration, i.e., $i = 5, j = 3$. We notice an overlap of three array elements, indicating a significant data reuse resulting from temporal locality of data access. In general, out of the L array elements accessed in the innermost loop, we have already accessed $L - 1$ elements in the previous j iteration. A suggested power optimization here is that,



```
for (i = 0; i < N; i++) {
   for (j = 0; j < N-L; j++) {
      b [i][j] = 0;
      for (k = 0; k < L; k++)
         b [i][j] += a [i][j+k];
   }
}
```

```
for (i = 0; i < N; i++) {
   for (j = 0; j < N-L; j++) {
      b_buf = 0;
      a_buf [(j+L-1)%L] = a [j+L-1];
      for (k = 0; k < L; k++)
         b_buf += a_buf [(j+k)%L];
      b[i][j] = b_buf;
   }
}
```

Fig. 4.36 Memory Customization Example (**a**) Original Loop (**b**) Default memory architecture (**c**) 'a' elements accessed in inner loop for i = 5, j = 2 (**d**) 'a' elements accessed in inner loop for i = 5, j = 3 (**e**) Modified loop (**f**) Customized memory architecture. Only the relatively smaller $M1$ is accessed in the inner loop, saving power

we can instantiate a small buffer *a_buf* of size L words which would store the last L elements of a accessed in the previous j-iteration. Since L is much smaller than N, we would be accessing data only from the much smaller $M1$ memory module shown in Fig. 4.36(f) in the innermost loop. Since the energy cost for access from $M1$ is expected to be much smaller than that due to access from *Mem*, we can expect a significant energy saving for the entire loop nest. The modified loop is shown in Fig. 4.36(e).

An explicit physical partitioning of the logical memory space could also be performed based on the dynamic profile of memory address references. This could be useful when a static analysis of the array references is difficult either due to complex conditionals in the specification, or due to data dependence. The frequency of memory references for an application is generally not uniformly distributed over the address space – certain parts of the memory are likely to be more heavily accessed than others. Figure 4.37(a) shows an example of a memory access frequency distribution over the address space. The accesses could be logically grouped into three windows of different characteristics – the first 256 addresses have a relatively high access frequency; the next 2048 elements have a low access frequency; and the final 1024 elements have a medium frequency. Such variations in access distributions could occur in typical code because different data arrays are accessed in different ways. Based on the above grouping, the memory space could be partitioned into three physical modules. Figure 4.37(b) shows a default memory architecture in which all memory accesses are made to one large memory module storing the entire address space. This could, however, lead to unnecessarily high memory access energy because every access would be made to the large memory. Figure 4.37(c)

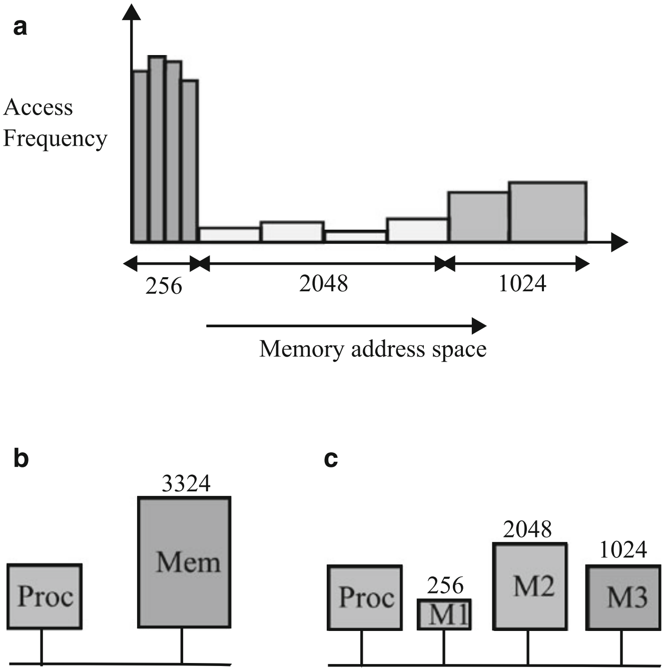


Fig. 4.37 Memory Partitioning. (**a**) Distribution of access frequency over memory address space. (**b**) Default memory architecture. (**c**) Partitioned memory. The most frequent accesses are made to the smaller $M1$, saving power

shows an example memory partitioning with the three logical address ranges identified above mapped to three separate physical modules. This partitioning ensures that the high frequency accesses are restricted to the relatively smaller memory module, thereby leading to a significant energy saving over the default architecture.

4.7 Reducing Address Bus Switching

The memory address bus is typically long because of several reasons. The memory core might have been separately obtained and instantiated, and it may not be possible to physically place it next to the address generation logic. The memory may be serving several units which are independently generating addresses and accessing data. Hence, bits switching on the address lines lead to significant power dissipation because of the large capacitance associated with each line. This provides an important power optimization opportunity – power can be reduced by reducing the total number of bits switching on the memory address lines. This reduction can be effected by two broad approaches: (i) encoding the address lines; and (ii) transforming the data layout.

4.7.1 Encoding

The sequence of bits appearing over the memory address bus can be changed by, in general, inserting an *encoder* close to the address generation block, and a *decoder* near the memory in such a way that the same addresses are generated and seen by the memory as before, but the sequence of signals appearing at the address bus is modified. The difference is illustrated in Fig. 4.38(a) and (b). The encoder and

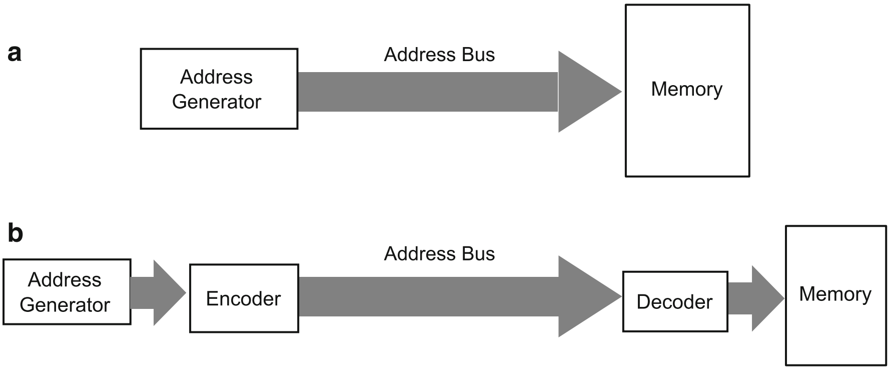
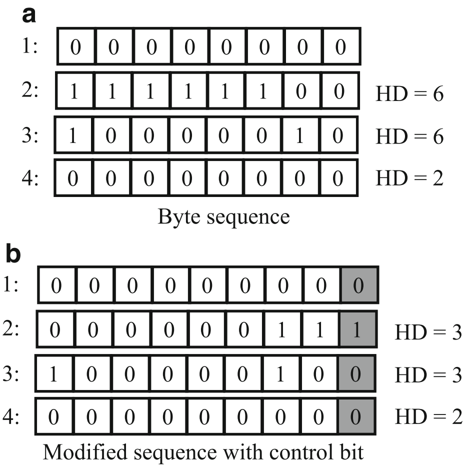


Fig. 4.38 Address Bus Encoding. (**a**) Original architecture (**b**) Encoder and decoder for address bus. The objective of encoding is to have lower activity on the encoded address bus at the expense of a small encoding and decoding power



a

1:	0	0	0	0	0	0	0	0	
2:	1	1	1	1	1	1	0	0	HD = 6
3:	1	0	0	0	0	0	1	0	HD = 6
4:	0	0	0	0	0	0	0	0	HD = 2

Byte sequence

b

1:	0	0	0	0	0	0	0	0	0	
2:	0	0	0	0	0	0	1	1	1	HD = 3
3:	1	0	0	0	0	0	1	0	0	HD = 3
4:	0	0	0	0	0	0	0	0	0	HD = 2

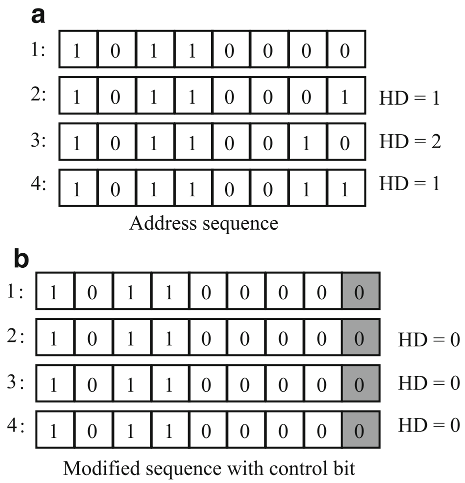
Modified sequence with control bit

Fig. 4.39 Bus Invert Coding. (**a**) Original sequence – total bits switching = 14 (**b**) Modified sequence – control bit = 1 indicates the data bits should be inverted. Total bits switching = 8

decoder logic blocks incur additional power overhead, but the power saved by reduced switching on the high-capacitance memory address lines is expected to be much larger. We study two very simple and effective encoding techniques in this section.

The *Bus-invert* encoding technique attempts to minimize the Hamming Distance between successive words appearing on the address bus [34]. An extra bit is added to the memory bus, indicating to the decoder whether or not the address word should be bitwise inverted. The choice of whether to invert or not is made depending on which option reduces the hamming distance between the current and previous words. This is illustrated in Fig. 4.39, with four successive addresses appearing on an 8-bit address bus. The four successive address values appearing on the address bus have hamming distances 6, 6, and 2, making a total of 14 bits switching. The bus-invert encoding mechanism adds an auxiliary bit to the address bus (shown in grey), making the width 9 bits. Noticing that 6 out of the 8 bits have switched in the second word, we decide to complement the address bits in the second word. The fact that the bits are inverted is transmitted by making the auxiliary bit ‘1’. This causes a total hamming distance of 3 between the first two words (two bits in the address word, and one due to the auxiliary bit). Comparing the second transmitted word with the third word, we notice a hamming distance of just 2, so the word is sent as is, with the auxiliary bit set to ‘0’ (indicating that the word is normal, not complemented). Thus, whenever the number of bits switching is more than half the bus width, we can send the complemented bits, thereby ensuring that no more than half the bits in the bus will change from one transaction to the next. The total number of bits switching in the encoded bus is 8, as opposed to 14 in the original bus. The encoding is a general mechanism and is not address-bus specific.

An encoding scheme that is specifically tailored to the typical behavior of memory address buses is the *T0* encoding. It exploits the general observation that often, the address sequence generated on the instruction memory address bus of



a

1:	1	0	1	1	0	0	0	0	
2:	1	0	1	1	0	0	0	1	HD = 1
3:	1	0	1	1	0	0	1	0	HD = 2
4:	1	0	1	1	0	0	1	1	HD = 1

Address sequence

b

1:	1	0	1	1	0	0	0	0	0	
2:	1	0	1	1	0	0	0	0	0	HD = 0
3:	1	0	1	1	0	0	0	0	0	HD = 0
4:	1	0	1	1	0	0	0	0	0	HD = 0

Modified sequence with control bit

Fig. 4.40 T0 Encoding. (**a**) Original sequence – total bits switching = 4 (**b**) Modified sequence – control bit = 1 indicates that the previous value should be incremented. Total bits switching = 0

a processor tends to have consecutive values because of spatial locality of reference in the instruction stream. The T0 code adds an extra line to the bus, which is used to indicate whether the next address is consecutive and is generated by incrementing the previous one [4]. This is illustrated in Fig. 4.40. The original sequence of four address has consecutive values, with the hamming distances between words being 1, 2, and 1, giving a total of 4 bits switching for the sequence (Fig. 4.40(a)). In T0 encoding the second address repeats the first, with the extra control bit being ‘0’, indicating that the the previous address should be incremented to generate the new one. This continues for the remaining words, with the decoder expected to increment the previous address to generate the new one as long as the control bit is ‘0’. This scheme may lead to near zero transitions in the steady state when there is a sequence of consecutive addresses. In addition to short consecutive sequences occurring in the instruction address sequence, this also happens during array data accesses in loops.

4.7.2 Data Layout

An orthogonal transformation to address encoding is to rearrange the data layout in memory so that the generated address sequence has lower switching when the data is accessed. Such layout transformations apply more to data memory than instructions.

Figure 4.41 shows an example of a simple data layout transformation that helps bring down the memory address bus switching. An example code is shown in Fig. 4.41(a), and its corresponding data memory access sequence is shown in Fig. 4.41(b). We notice that the memory address alternates between two regions of the memory corresponding to arrays a and b. This usually results in a large number of address bits switching between every consecutive pair of address words (with

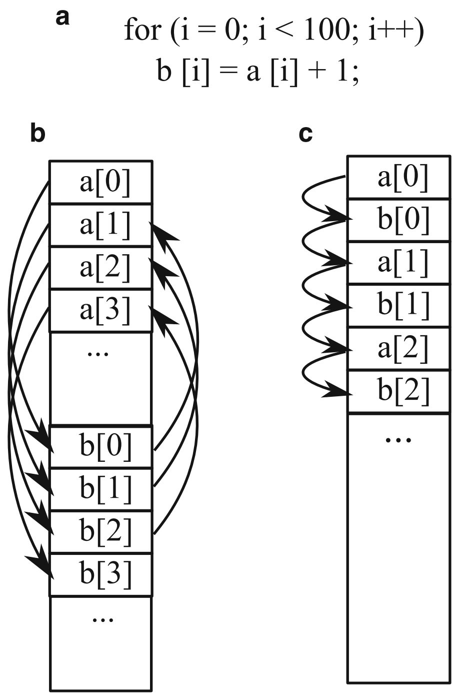
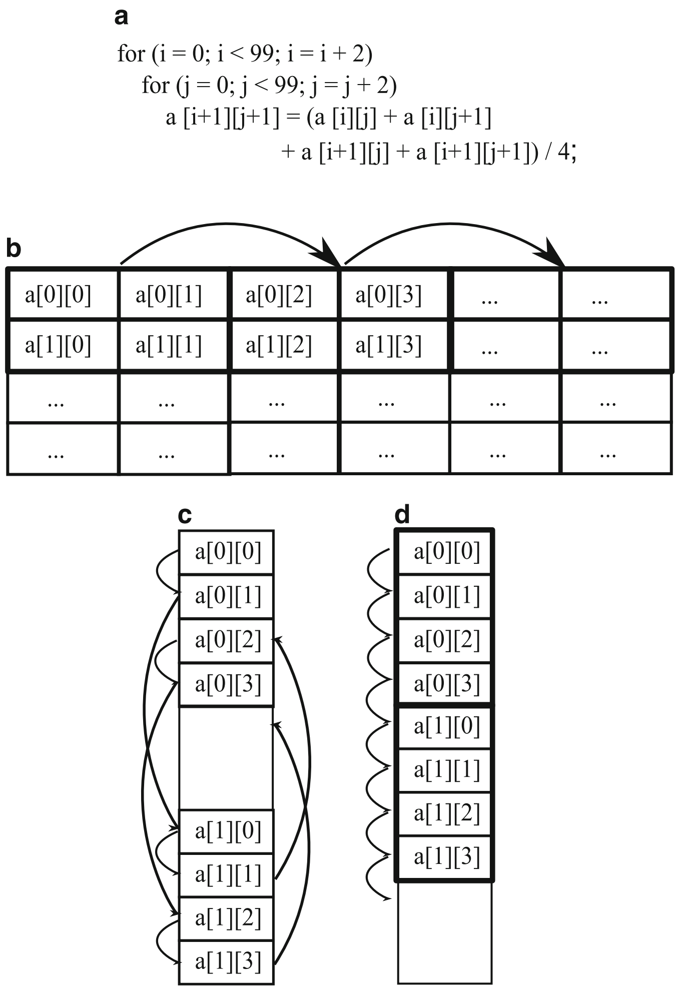


Fig. 4.41 Data Interleaving. (**a**) Behavior (**b**) Original address sequence (**c**) Address sequence when arrays are interleaved. The interleaved address sequence usually has lower total switching, and is amenable to other encoding strategies such as gray code

the exception of the specific case where the corresponding data elements of equal width are separated by an exact power of two, in which case only one bit would be flipping). Since the sequence is deterministic, we can perform a simple transformation of interleaving the elements of the two arrays, as shown in Fig. 4.41(c). This causes the address sequence to be consecutive, which is much better behaved in terms of bit switching, and can then be exploited by other encoding and decoding mechanisms such as Gray Code or T0.

A more complex data transformation is shown in Fig. 4.42. In the example of Fig. 4.42(a), the two-dimensional array *a* is accessed in a *tiled* pattern visually depicted in Fig. 4.42(b). Using the standard storage conventions of row-major and column-major storage for multi-dimensional arrays, we see that the address sequence incurs large hops even within a tile, since we have more than one row and column accessed in each tile. The sequence for row-major storage is shown in Fig. 4.42(c). Again, the predictability of the behavioral array reference pattern can be used to use a more custom *tile-based* mapping to store array data in memory. Figure 4.42(d) shows the tiles laid out in memory in a manner that avoids the large hops in the memory address bus.

Detailed discussions on encoding and data transformations for reducing address bus power can be found in [4, 29]. Before deciding on the applicability of such encoding and transformations to a specific system design scenario, it is important to perform a careful cost-benefit analysis.



```
for (i = 0; i < 99; i = i + 2)
   for (j = 0; j < 99; j = j + 2)
      a [i+1][j+1] = (a [i][j] + a [i][j+1]
                    + a [i+1][j] + a [i+1][j+1]) / 4;
```

Fig. 4.42 Tile-based data storage. (**a**) Behavior (**b**) Tiled memory access pattern (**c**) Address sequence using row-major storage (**d**) Address sequence using tile-based storage. Access pattern is more regular and shows lower total switching in tile-based storage

- The address bus encoding decision would depend on the actual address bus lengths in the system. If the on-chip bus has a relatively small length, the area and power overhead incurred in encoding and decoding circuitry may not be worthwhile.
- The actual memory architecture and address protocol may also influence the applicability of such optimizations, a good example being a DRAM. In DRAMs, the actual address is time-multiplexed into row and column addresses, using half the number of address bits. This obviously requires changes to the simple encoding techniques discussed above based on hamming-distance between complete

addresses – here, we need to worry about the hamming distance between the row- and column address as well.

- Address bus hamming distance is not the only metric of importance in determining total energy. It is possible that a hamming distance of 1 refers to a physically distant memory location if the switch happens in the higher order address bits. This may cause a switch in the *DRAM memory page*, which will lead to additional power costs in the DRAM and associated changes in the access protocol FSM that should be accounted for.
- A similar cost analysis has to be performed for data transformations targeting the memory address bus. Data transformations are more global in their effect – they go beyond the specific loop for which the transformation was derived. Transformations to the same arrays in different loops might be conflicting – they may be useful in one and harmful in another. Data transformations need to be performed after analyzing their effect on all sections of code where the data is accessed.

4.8 DRAM Power Optimization

As the amount of memory in computer systems continues to grow, the off-chip memory – the DRAM sub-system – increasingly consumes larger amounts of power, even though the CPU is designed against a tight power budget. Newer generations of DRAMs have exposed power management controls so that external controllers can implement aggressive optimization policies to reduce DRAM power. The most common such feature has been the introduction of low power modes in different DRAM states. These could be one of the following:

- *Active:* The DRAM could be moved to an *Active Power-down* mode. In this "shallow" low-power mode, one or more DRAM banks is open, and it is relatively fastest to bring the DRAM back to accessing data from this mode.
- *Precharge:* In the *Precharge Power-down* mode, all banks are closed, and it takes more time to return to the active state where we can access data again. Power consumption is lower than in active power-down.
- *Self-refresh:* In the *Self-refresh Power-down* mode, the power consumption is the lowest, and it takes the maximum time to recover to the active state.

The transitions to and from the above DRAM low power states is expected to be explicitly performed by the memory controller. The architectural details and programming possibilities continue to evolve with newer generation of DRAMs.

The above power reduction opportunities have led to attention being focussed on optimizations through various mechanisms – starting from the level of memory controllers that directly interface to DRAMs, all the way to page allocation policies in operating systems. DRAM power management functions can be performed by many different entities: the DRAM controller can use prediction techniques; the compiler can analyze the source code and insert explicit instructions to change DRAM power state; and the operating system scheduler can also monitor activity on the DRAM and make the management decisions. Since the operating system is able to simultaneously monitor activities of different processes, it may discover reference patterns not visible to the compiler.

It is obvious that some variant of the standard predictive shutdown policy generally useful in several other low power controller strategies – predict the future idle time based on past behavior, and switch to low power mode if there is an expectation of power saving – is also applicable in the DRAM context (see Section 5.1.2 for an application of the same principle to voltage scaling decisions based on past CPU utilization). Basically, the controller's policy should ensure that the overhead of transitioning to and from the low power state is less than the expected power saving [7]. The choice of the controller policy would depend on the amount of performance penalty we are willing to tolerate. This is achieved by setting specific threshold parameters for every power state of the DRAM. If the chip has not been accessed for a time period beyond the threshold, then we can move the DRAM to the next low power state [24].

The data migration policy discussed in Section 4.5 – where data objects are grouped into a smaller number of memory modules so that others can be set to sleep mode – is an example of a high-level data organization concept that also applies to DRAM. The default page allocation of the operating system, which effectively does a random assignment of memory pages across the different memory chips present in the system, can be made power aware by exploiting the same data consolidation idea introduced in Section 4.5 – frequently accessed memory pages are migrated to a common memory chip so as to improve the possibility of power saving through moving unused chips to low power mode [24].

Finally, attempts can be made to bring successive DRAM accesses closer in time so that idle periods can be made artificially longer. Bringing two DRAM accesses together is usually achieved by delaying the first access, which may incur a performance penalty.

4.9 Summary

The storage and retrieval of large amounts of data and instructions in modern electronic and computer systems make the memory subsystem an important target component for power optimization. Memory related power reduction spans a broad spectrum across several levels of abstraction. Circuit and architecture level opportunities have been identified for trading off a small performance penalty for significant savings in power dissipation in processor caches. Scratch pad memories help reduce system power by avoiding expensive tag lookups associated with caches. Since SoCs permit flexible on-chip memory architectures, estimation based exploration can help determine the best memory configurations for an application scenario. Techniques such as encoding and data layout transformation could be used to reduce power dissipation on high-capacitance memory address buses.

The chapter covered some of the basic techniques in each of the categories described above. The reader is encouraged to look up the references indicated at various places in the chapter to obtain an idea of the more advanced proposals in the memory power optimization field. The area continues to draw significant interest from researchers as systems move in the direction of higher complexity. For example, as next generation DRAMs evolve to include more power controls, we can expect associated power optimization proposals to exploit them. Advanced transformations such as compression also help reduce power by reducing off-chip memory traffic and providing similar performance with smaller memory and cache sizes.

References

1. Bajwa, R.S., Hiraki, M., Kojima, H., Gorny, D.J., Nitta, K., Shridhar, A., Seki, K., Sasaki, K.: Instruction buffering to reduce power in processors for signal processing. IEEE Transactions on VLSI Systems **5**(4), 417–424 (1997)
2. Bellas, N., Hajj, I., Polychronopoulos, C.: Using dynamic cache management techniques to reduce energy in a high-performance processor. In: International symposium on low power electronics and design, pp. 64–69. San Diego, USA (1999)
3. Bellas, N., Hajj, I.N., Polychronopoulos, C.D., Stamoulis, G.: Architectural and compiler techniques for energy reduction in high-performance microprocessors. IEEE Transactions on VLSI Systems **8**(3), 317–326 (2000)
4. Benini, L., Micheli, G.D.: System level power optimization: Techniques and tools. ACM Transactions on Design Automation of Electronic Systems **5**(2), 115–192 (2000)
5. Chang, Y.J.: An ultra low-power tlb design. In: DATE '06: Proceedings of the conference on Design, automation and test in Europe, pp. 1122–1127. European Design and Automation Association, 3001 Leuven, Belgium, Belgium (2006)
6. Delaluz, V., Sivasubramaniam, A., Kandemir, M., Vijaykrishnan, N., Irwin, M.J.: Scheduler-based dram energy management. In: Design Automation Conference. New Orleans, USA (2002)
7. Fan, X., Ellis, C., Lebeck, A.: Memory controller policies for dram power management. In: International symposium on low power electronics and design, pp. 129–134. Huntington Beach, USA (2001)
8. Flautner, K., Kim, N.S., Martin, S., Blaauw, D., Mudge, T.: Drowsy caches: simple techniques for reducing leakage power. In: International symposium on computer architecture, pp. 240–251. Anchorage, USA (2002)
9. Garey, M.R., Johnson, D.S.: Computers and Intractibility – A Guide to the Theory of NP-Completeness. W.H. Freeman (1979)
10. Ghose, K., Kamble, M.B.: Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In: International symposium on low power electronics and design, pp. 70–75. San Diego, USA (1999)
11. Hasegawa, A., Kawasaki, I., Yamada, K., Yoshioka, S., Kawasaki, S., Biswas, P.: SH3: High code density, low power. IEEE Micro **15**(6), 11–19 (1995)
12. Hennessy, J.L., Patterson, D.A.: Computer Architecture – A Quantitative Approach. Morgan Kaufman, San Francisco, CA (1994)
13. Hu, J.S., Nadgir, A., Vijaykrishnan, N., Irwin, M.J., Kandemir, M.: Exploiting program hotspots and code sequentiality for instruction cache leakage management. In: International symposium on low power electronics and design, pp. 402–407. Seoul, Korea (2003)
14. Inoue, K., Ishihara, T., Murakami, K.: Way-predicting set-associative cache for high performance and low energy consumption. In: International symposium on low power electronics and design, pp. 273–275. San Diego, USA (1999)

15. Inoue, K., Moshnyaga, V.G., Murakami, K.: A history-based I-cache for low-energy multimedia applications. In: International symposium on low power electronics and design, pp. 148–153. Monterey, USA (2002)
16. Janapsatya, A., Parameswaran, S., Ignjatovic, A.: Hardware/software managed scratchpad memory for embedded systems. In: Proceedings of the IEEE/ACM International Conference on Computer Aided Design (2004)
17. Juan, T., Lang, T., Navarro, J.J.: Reducing tlb power requirements. In: ISLPED '97: Proceedings of the 1997 international symposium on Low power electronics and design, pp. 196–201. ACM, New York, NY, USA (1997). DOI http://doi.acm.org/10.1145/263272.263332
18. Kadayif, I., Sivasubramaniam, A., Kandemir, M., Kandiraju, G., Chen, G.: Generating physical addresses directly for saving instruction tlb energy. In: MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture, pp. 185–196. IEEE Computer Society Press, Los Alamitos, CA, USA (2002)
19. Kandemir, M., Ramanujam, J., Irwin, M.J., Vijaykrishnan, N., Kadayif, I., Parikh, A.: Dynamic management of scratch-pad memory space. In: ACM/IEEE Design Automation Conference, pp. 690–695 (2001)
20. Kaxiras, S., Hu, Z., Martonosi, M.: Cache decay: exploiting generational behavior to reduce cache leakage power. In: International symposium on computer architecture, pp. 240–251. Goteberg, Sweden (2001)
21. Kim, S., Vijaykrishnan, N., Kandemir, M., Sivasubramaniam, A., Irwin, M.J., Geethanjali, E.: Power-aware partitioned cache architectures. In: International symposium on low power electronics and design, pp. 64–67. Huntington Beach, USA (2001)
22. Kin, J., Gupta, M., Mangione-Smith, W.H.: The filter cache: an energy efficient memory structure. In: International symposium on microarchitecture, pp. 184–193. Research Triangle Park, USA (1997)
23. Ko, U., Balsara, P.T., Nanda, A.K.: Energy optimization of multi-level processor cache architectures. In: International symposium on low power design, pp. 45–49. New York, USA (1995)
24. Lebeck, A.R., Fan, X., Zeng, H., Ellis, C.: Power aware page allocation. SIGOPS Oper. Syst. Rev. **34**(5), 105–116 (2000). DOI http://doi.acm.org/10.1145/384264.379007
25. Lyuh, C.G., Kim, T.: Memory access scheduling and binding considering energy minimization in multi-bank memory systems. In: Design automation conference, pp. 81–86. San Diego, USA (2004)
26. Malik, A., Moyer, B., Cermak, D.: A low power unified cache architecture providing power and performance flexibility. In: International symposium on low power electronics and design, pp. 241–243. Rapallo, Italy (2000)
27. Manne, S., Klauser, A., Grunwald, D.C., Somenzi, F., Somenzi, F.: Low power tlb design for high performance microprocessors. Tech. rep., University of Colorado (1997)
28. Min, R., Jone, W.B., Hu, Y.: Location cache: A low-power l2 cache system. In: International symposium on low power electronics and design, pp. 120–125. Newport Beach, USA (2004)
29. Panda, P.R., Catthoor, F., Dutt, N.D., Danckaert, K., Brockmeyer, E., Kulkarni, C., Vandercappelle, A., Kjeldsberg, P.G.: Data and memory optimization techniques for embedded systems. ACM Transactions on Design Automation of Electronic Systems **6**(2), 149–206 (2001)
30. Panda, P.R., Dutt, N.D., Nicolau, A.: Local memory exploration and optimization in embedded systems. IEEE Transactions on Computer Aided Design **18**(1), 3–13 (1999)
31. Panda, P.R., Dutt, N.D., Nicolau, A.: Memory Issues in Embedded Systems-On-Chip: Optimizations and Exploration. Kluwer Academic Publishers, Norwell, MA (1999)
32. Panda, P.R., Dutt, N.D., Nicolau, A.: On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems. ACM Transactions on Design Automation of Electronic Systems **5**(3), 682–704 (2000)
33. Panwar, R., Rennels, D.: Reducing the frequency of tag compares for low power i-cache design. In: International symposium on low power design, pp. 57–62. New York, USA (1995)
34. Stan, M.R., Burleson, W.P.: Bus-invert coding for low power I/O. IEEE Transactions on VLSI Systems **3**(1), 49–58 (1995)

35. Steinke, S., Wehmeyer, L., Lee, B., Marwedel, P.: Assigning program and data objects to scratchpad for energy reduction. In: Design Automation and Test in Europe, pp. 409–417. Paris, France (2002)
36. Su, C.L., Despain, A.M.: Cache design trade-offs for power and performance optimization: a case study. In: International Symposium on Low Power Design, pp. 63–68. New York, NY (1995)
37. Udayakumaran, S., Dominguez, A., Barua, R.: Dynamic allocation for scratch-pad memory using compile-time decisions. ACM Transactions on Embedded Computing Systems **5**(2), 472–511 (2006)
38. Zhang, C., Vahid, F., Yang, J., Najjar, W.: A way-halting cache for low-energy high-performance systems. In: International symposium on low power electronics and design, pp. 126–131. Newport Beach, USA (2004)
39. Zhang, W., Hu, J.S., Degalahal, V., Kandemir, M., Vijaykrishnan, N., Irwin, M.J.: Compiler directed instruction cache leakage optimization. In: International symposium on microarchitecture, pp. 208–218. Istanbul, Turkey (2002)

Chapter 5
Power Aware Operating Systems, Compilers, and Application Software

What does a compiler have to do with power dissipation? A compiler is a piece of system software that parses a high level language, performs optimizing transformations, and finally generates code for execution on a processor. On the surface, it seems very far removed from an electrical phenomenon like power dissipation. Yet, it was not long before the two got inextricably linked. The involvement of the compiler along with the processor architecture in the design space exploration loop of application specific systems (ASIPs) might have eased the transition. In this scenario, compiler analysis can actually influence the choice of architectural parameters of the final processor. Clearly, if a low power system consisting of an application running on a processor is desired, the selected processor architecture has to work in tandem with the compiler and application programmer – an architectural feature is useless if it is not properly exploited by the code generated by a compiler or written by a programmer. Low power instruction encoding is an example optimization that features the compiler in a central role with the explicit role of reducing power. In an ASIP, the opcode decisions need not be fixed, and could be tuned to the application. Since the compiler has an intimate knowledge of the application, it could anticipate the transition patterns between consecutive instructions from the program layout and suggest an encoding of instructions that reduces switching power arising out of the fetch, transmission, and storage of sequences of instructions. Modern compiler designers are investigating the development of power awareness in a more direct way in general purpose processor systems, not just ASIPs. The role of the compiler and application programmer grows along with the concomitant provision of hooks and control mechanisms introduced by the hardware to support high level decision making on power-related issues.

The operating system has a very direct role in power management of a computer system, since it has the important responsibility of monitoring and controlling every system resource. Of course, the hardware resource itself might be designed to save power when conditions are favorable – for example, a memory device can shift to low power mode when it is inactive for a long period of time. However, system level power management can be more aggressive if the operating system plays an active role in addition to power efficiencies built into individual resources. For example, an individual resource may require an accurate prediction of future activity, in order to make good power management decisions. Since the operating system also assumes

P.R. Panda et al., *Power-efficient System Design*, DOI 10.1007/978-1-4419-6388-8_5,

the responsibility for task allocation on resources, it may have that vital dynamic information using which it can inform the resource whether any significant workload is likely to be scheduled on it in the near future. It is clear that aggressive power optimizations can take place when there is a meaningful collaboration between the operating system and the resources it manages.

In this chapter we study some recent work in the area of power aware operating systems, compilers, and application software. This continues to be an important research area and we can expect exciting new problems and solutions in the days ahead.

5.1 Operating System Optimizations

An operating system is very well placed to make intelligent run-time power management decisions because it is best suited to keep track of the dynamic variation of the status of the different resources under its supervision. Before studying the power optimization policies implemented by an OS, it is instructive to look at the component-wise break-up of the power dissipation on a typical computer. As expected, the total power dissipation, and the relative power dissipated in the individual components, vary depending on the benchmark/application domain.

Figure 5.1 gives a comparison of the total system power of an IBM Thinkpad R40 laptop with a 1.3 GHz processor, 256 MB memory, 40 GB hard drive, optical drives, wireless networking, and a 14.1” screen, when it is subject to workloads arising out of different classes of benchmark applications [30]. The idle system dissipates 13 W,

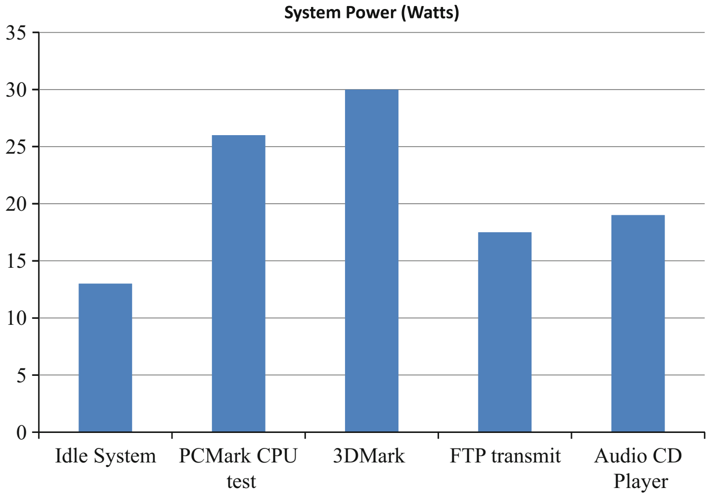


Fig. 5.1 Laptop power dissipation for different benchmarks. Idle power is 13 W. The benchmarks vary widely in their power dissipation

while the benchmarks (*3DMark* – graphics benchmark; *CPUMark* – CPU intensive workload; *Wireless FTP* – file transfer over the wireless LAN card; and *Audio CD Playback*) dissipate between 17 W and 30 W, exhibiting a wide range.

Figures 5.2, 5.3, 5.4, 5.5, and 5.6 show the component-wise break-up of the total system power dissipation for the different benchmarks and idle state. In an idle system, the LCD display consumes a relatively large fraction of the power. For the CPU-intensive PCMark suite, the CPU was, as expected, the largest consumer of

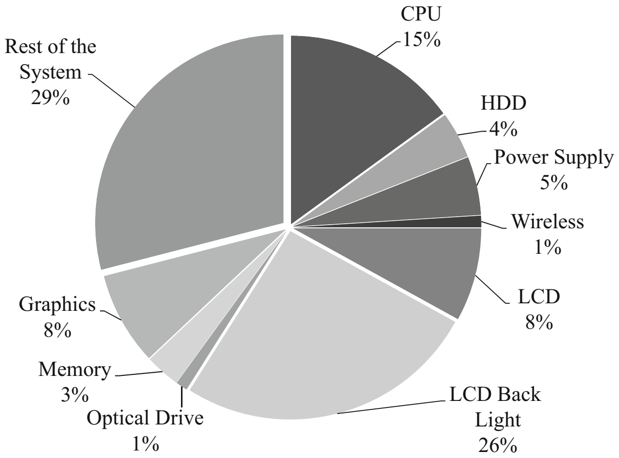


Fig. 5.2 Component-wise break-up of laptop power dissipation when the system is idle. In idle systems, the LCD display consumes a relatively large fraction (26%) of the total power

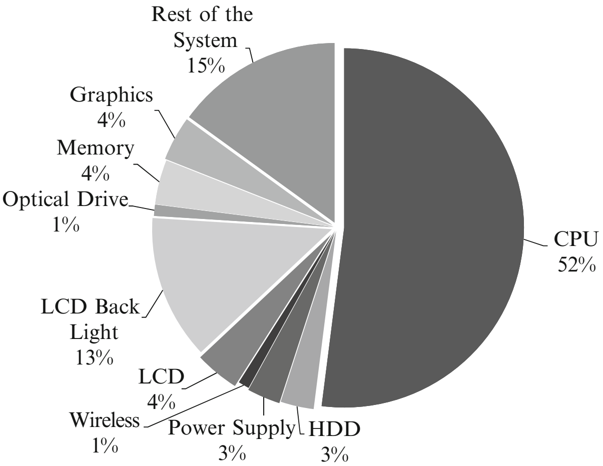


Fig. 5.3 Component-wise break-up of laptop power dissipation for CPUMark benchmark. This benchmark stresses the CPU, which consumes 52% of the total power

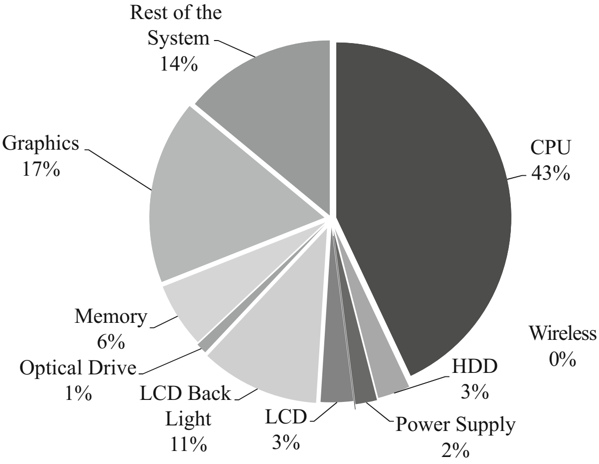


Fig. 5.4 Component-wise break-up of laptop power dissipation for 3DMark benchmark. The CPU consumes a large 43% of the system power

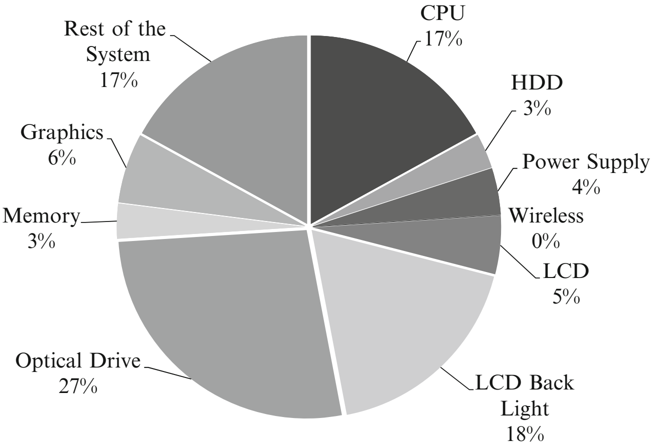


Fig. 5.5 Component-wise break-up of laptop power dissipation for Audio CD player benchmark. The optical drive was the main power consumer, accounting for 27%

power. The 3DMark suite is also CPU intensive, and the CPU power accounted for a huge 43%. FTP showed a relatively large power dissipation in the wireless card, drawing power comparable to the CPU. In the Audio CD playback, the optical drive was the main power consumer, with its power exceeding even the CPU power, because the drive was running at full speed throughout the playback period. The study shows that the power distribution among the system components depends on the type of computational and data transfer demands placed on the individual resources. The CPU is usually among the heaviest power users.

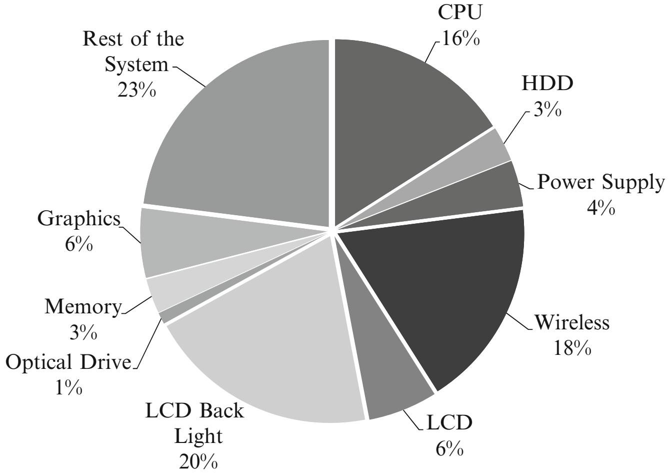


Fig. 5.6 Component-wise break-up of laptop power dissipation for wireless FTP benchmark. The wireless card dissipates 18% of the system power

Table 5.1 Hard disk drive states [30]

HDD power state	Power Consumption
Idle	0.575 W
Standby	0.173 W
Read	2.78 W
Write	2.19 W
Copy	2.29 W

Table 5.2 Power consumption variation of LCD display with brightness level [30]

LCD brightness level	Power Consumption
1	0.6 W
2	0.8 W
3	1.1 W
4	1.3 W
5	1.6 W
6	2.0 W
7	2.7 W
8	3.4 W

The significant difference in power consumption of individual components across the various benchmark applications can be attributed to the different power characteristics of the devices based on the usage pattern of the application. Tables 5.1, 5.2, 5.3, and 5.4 show the power drawn by the hard disk drive, LCD display, Wireless LAN card, and CD drive, in the different *power states* of the respective devices. The devices exhibit a significant dynamic power dissipation range, depending on the state of activity. For example, the wireless LAN card draws 22 times as much power when it is transmitting, compared to when it is idle. Similarly, there is a 7:1

Table 5.3 Wireless LAN card states [30]

Wireless LAN Card states	Power Consumption
Power saver (idle)	0.14 W
Base (idle)	1.0 W
Transmit	3.2 W at 4.2 Mb/s
Receive	2.55 W at 2.9 Mb/s

Table 5.4 Optical drive states [30]

Optical drive state	Power Consumption
Initial spin up	3.34 W
Steady spin	2.78 W
Reading data	5.31 W

power dissipation ratio between the highest and lowest brightness levels of the LCD display, and a 16:1 ratio between active and idle power for the hard disk.

5.1.1 Advanced Configuration and Power Interface (ACPI)

In a typical computing system with several input/output devices, processing units, memory devices, etc., working in unison, it is very unlikely that all of them would be active for the entire duration of system operation. For example, a modem may be active only when applications running on the system request a network access, and is inactive for the rest of the time. It is observed that a significant amount of power is dissipated during these spells of inactivity owing to the following reasons:

- With technology scaling, leakage power has become a significant portion of the total power consumption of an electronic circuit. For example, 40% of the 110W of power consumed by a 90nm Pentium 4 is actually due to the leakage current [35].
- Devices such as display, waste power in doing redundant work. Displays are designed such that the contents of the framebuffer are refreshed periodically on to the display device. Even when the system is idle and the contents of framebuffer are not expected to change, the display is periodically refreshed, resulting in waste of power.

It would appear that an ideal solution to counter leakage power would be to activate the device only when it is working. But this is not always feasible, since the switching time from *on* to *off* and vice-versa could affect system performance. Hence the devices are generally designed to work at different operating points called the *power modes* that represent trade-offs between performance and power. Depending on the requirements of the applications running on the system, the mode of device operation is dynamically selected and modified. The policies that govern the switching of operating point of a device are called the power management policies. Power management of devices could be implemented in two ways:

- in the firmware of the device and controlled by the driver of the device; and
- in the operating system.

Operating System directed Power Management (OSPM) is becoming popular in modern systems due to the following advantages over device level implementations.

- Implementation of power management in the OS makes it platform independent.
- The limitation of implementing complex power management strategies in the BIOS of the devices can be overcome.
- Algorithms common to power management of several devices can be implemented only once, thus decreasing the development cost.

Now that the power management policies are implemented in the OS, standard interfaces between the OS and device drivers are necessary for smooth operation. Advanced Configuration and Power Interface (ACPI) is the specification of a common standard for OS controlled device configuration and power management [16]. This standard was initially developed by Intel, Microsoft, and Toshiba, with Hewlett-Packard and Phoenix being involved in the later evolution.

Let us examine the ACPI standard in some detail. Figure 5.7 shows the architecture of a system using ACPI for power management. The operating system communicates with the ACPI stack through software and drivers. The ACPI layer acts as an interface between the OS and the device with the help of three main components: (i) ACPI tables; (ii) ACPI bios; and (III) ACPI registers.

ACPI tables contain definition blocks that describe the ACPI managed devices. The definition includes data and machine independent byte-code that performs device configuration and management.

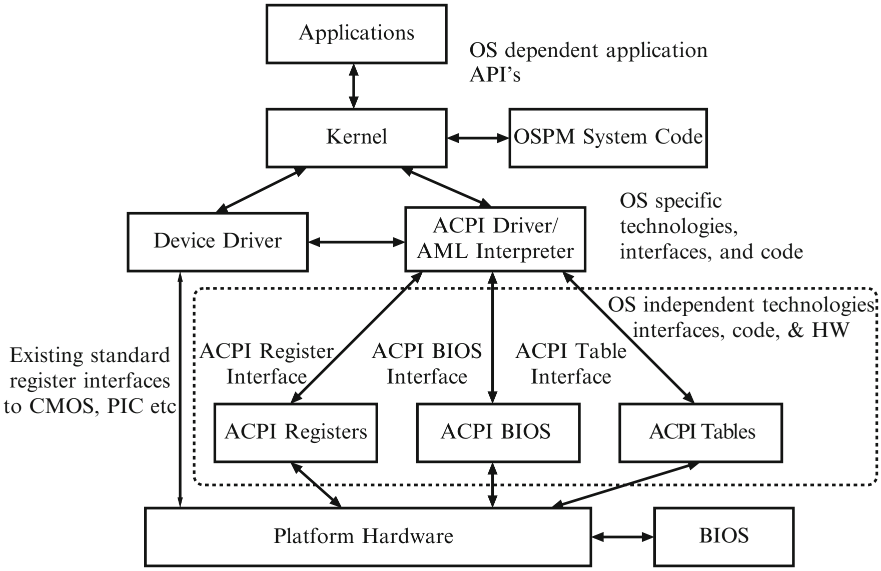

Applications
OS dependent application API's
Kernel
OSPM System Code
Device Driver
ACPI Driver/ AML Interpreter
OS specific technologies, interfaces, and code
OS independent technologies interfaces, code, & HW
Existing standard register interfaces to CMOS, PIC etc
ACPI Register Interface
ACPI BIOS Interface
ACPI Table Interface
ACPI Registers
ACPI BIOS
ACPI Tables
Platform Hardware
BIOS

Fig. 5.7 ACPI system architecture

ACPI BIOS is responsible for low-level management operations of the device. It contains code to assist in booting the system and switch the operation mode of the device. Different power modes of a system are described in the following section.

ACPI registers are a set of hardware registers that help in configuration and management of the device. These registers are accessed through the byte-code stored in the device-specific part of the ACPI tables.

In an ACPI based system, on power-up, the ACPI BIOS is loaded prior to the OS and the ACPI tables are loaded into memory. Since the memory requirement of these tables is much more than what a BIOS memory could accommodate, the ACPI BIOS allocates space in the physical memory of the system. When the ACPI-aware OS kernel starts its operation, it searches the BIOS memory area to obtain the address of the ACPI tables in the physical memory. All ACPI operations, excluding a few BIOS functions, are performed in the OS by interpreting the machine-independent ACPI Machine Language (AML) byte-code present in the ACPI tables.

5.1.1.1 Power Modes

ACPI defines various power states in which the entire system and also individual devices in the system could be operating. These states are enumerated in Tables 5.5, 5.6, 5.7, and 5.8.

The *global states* apply to system as a whole, and are visible to the user. The G0 state ("working") is the normal active state in which user threads are executed.

Table 5.5 Global States that define the power mode of the entire system

State	Description
G0	*Working*
G1	*Sleeping* (subdivided into states S1 to S4) • S1 – All CPU caches are flushed and CPU(s) stop executing instructions. Full context is maintained in RAM. Power to RAM and CPU(s) is maintained. • S2 – CPU is powered off. Processor context and cache context are not maintained but RAM is maintained. • S3 – Only RAM remains powered, hence all system context is lost. (commonly referred as *Standby* or *Sleep*) • S4 – Data in RAM is flushed to hard disk and the system is powered down. (referred to as *Hibernation*) Reboot is required to wake up the system.
G2(S5)	*Soft off* In this state all the devices are also powered down along with CPU and caches. Some components in the system such as mouse and keyboard remain powered to wake up the system.
G3	*Mechanical off* The system is switched off except for the real time clock in the system that is powered by a small battery.

Table 5.6 Device Power States

State	Description
D0	Fully on.
D1,D2	Intermediate device dependent power states
D3	Powered off

Table 5.7 Processor Power States

State	Description
C0	*Fully on*.
C1	*Halt*. Processor does not execute any instructions, but can instantaneously return to execution
C2	*Stop-Clock*. Maintains application-visible state, but takes longer for wake-up.
C3	*Sleep*. Processor does not keep its cache, but maintains other state.

Table 5.8 Processor Performance States

State	Description
P0	Maximum power and frequency
P1	Less than P0, voltage and frequency scaled
Pn	Less than P(n-1), voltage and frequency scaled

System power consumption in this state is generally the highest. In the G1 state ("sleeping"), user threads are not executed and key components such as display are turned off to save power. However, the system can be moved to active state in a relatively small amount of time. In the G2 state ("soft off"), the system is mostly off, but some components remain "on" so that the system can "wake up" through signals received from an I/O device such as keyboard or mouse. Power consumed in this state is very low. In G3 state ("mechanical off") the system is turned off completely and draws near zero power, except for a clock powered by a small battery.

The global sleeping state is subdivided into several further levels S1 through S4, representing a finer grain power management. In the S1 state, the CPU caches are flushed and the CPU does not execute instructions, but can be moved to an active execution quickly. In the S2 state, the CPU is powered off, so the processor and cache contexts are lost, but the RAM is maintained. In the S3 state (commonly known as "standby" or "sleep"), the rest of the chip set is turned off, but the RAM is still valid. In the S4 state (commonly known as "hibernate"), the memory data is saved into the hard disk and the system is powered down. A reboot is required to wake the system up. The S5 state coincides with the G2 state.

The *device power states* D0 through D3 in ACPI apply to I/O devices connected to the system bus. These states are defined in a very generic manner, and some devices may not feature all the four states. In the D0 state, the device is "fully on" and consumes maximum power. The D1 and D2 states are low power states that are device dependent. In D3, the device is turned off and consumes no power. Re-initialization is necessary when the OS turns the device back3 on. Examples of device power states and the power consumed in each state are given in Tables 5.1, 5.2, 5.3, and 5.4, for the laptop experiment discussed above.

The *Processor power states* C0 through C3 represent various performance-power trade-offs in the processor in the global state G0 ("working"). In the C0 state ("fully on"), the processor executes normal instructions and consumes the highest power. In the C1 state ("halt"), the processor does not execute any instruction, but can immediately return to execution. In the C2 state ("stop clock"), the processor moves to a low power state, does not execute instructions, and takes longer to return to execution. The C3 state ("sleep") offers further improvements in power savings, with caches maintaining state but disallowing snooping.

Finally, the standard also defines the device and processor performance states within the respective "fully on" states D0 and C0. State P0 represents the maximum frequency for the CPU, which translates to maximum power consumption. Other states P1, P2, P3, etc., are defined, with decreasing power and associated performance. Dynamic voltage and frequency scaling (Section 5.1.2) is typically employed in the processors and devices to achieve the different power states.

5.1.2 Dynamic Voltage and Frequency Scaling

The basic dynamic power equation $P = CV^2Af$, where C is the load capacitance, V is the operating voltage, A is aggregate activity factor, and f is the operating frequency, shows the significant leverage possible by adjusting the voltage and frequency. It shows that if we can reduce voltage by some small factor, we can reduce power by the square of that factor. Furthermore, reducing supply voltage often slows transistors such that reducing the clock frequency is also required. The benefit of this is that within a given system, scaling supply voltage down now offers the potential of a cubic reduction in power dissipation. This process of reducing both the voltage and frequency of the processor dynamically (at run time) is called Dynamic Voltage and Frequency Scaling (DVFS). It is important to note here that while DVFS may reduce the instantaneous power cubically, the reduction on the total energy dissipated is quadratic. This is because, even though programs run at lower power dissipation levels, they run for longer durations.

Fundamentally, DVFS approaches exploit slack to achieve more power-efficient execution. The workload profile of applications is far from a constant; in fact, it may be highly dynamic. As a result, the processor need not be operating at the maximum performance (maximum voltage and frequency) all the time. There may be several opportunities to temporarily slow down the processor without any noticeable or adverse effects. For example, a CPU might normally respond to a user's command by running at full speed for 0.001 seconds, then waiting idle; running at one-tenth the speed, the CPU could complete the same task in 0.01 seconds, thereby saving power and energy without generating noticeable delay. Eventually DVFS is an approach that attempts to meet the seemingly conflicting goals of a responsive and intelligent device while maximizing battery life.

One of the most important decisions in implementing DVFS is the granularity at which to perform DVFS. The finest granularity at which DVFS is limited by the time it takes to switch the voltage and frequency of the processor. DVFS implementation requires a voltage regulator that is fundamentally different from a standard voltage regulator because it must also change the operating voltage for a new clock frequency [4,7,9]. This and other considerations result in high transition overhead for DVFS. This overhead is typically in the range of tens of micro seconds. In particular, the Intel XScale processor has a frequency switching time of $20\mu s$ [10,17,18]. To be able to profitably apply DVFS, and hide the penalty of voltage regulation, the granularity at which voltage and frequency are scaled should be at least 2 to 3 orders of magnitude higher, which is in the range of milliseconds. This falls more or less in the domain of operating system scheduling granularity. Consequently, most DVFS schemes have been incorporated into the OS scheduler and task management services.

The simplest application of DVFS algorithms is a history-based scheme, where we monitor the recent history to make a prediction about the immediately future. The *Past* algorithm is a simple strategy that divides time into *intervals* [45]. In each interval, the algorithm keeps track of what the CPU utilization was, and predicts that the utilization will remain unchanged in the next interval. This assumption is in keeping with system behavior in general – drastic changes in system load are relatively rare. The utilization is compared against a pre-defined *threshold*. If the utilization is below this threshold, then the system is slowed down by lowering the voltage. If the utilization is above the threshold, then the system is sped up by selecting a higher voltage. The strategy is illustrated in Fig. 5.8, with the threshold set at 80% utilization. In Fig. 5.8(a), a 70% utilization is observed in time interval t. The Past algorithm predicts a 70% utilization for interval $t + 1$, and slows down the system by stepping down the voltage. Similarly, in Fig. 5.8(b), a 90% utilization causes Past to step up the voltage. In order to prevent switching of voltages too frequently, the threshold can instead be defined as a range of utilizations, for example, between 75-85% in our example.

The Past algorithm is very simple, and suffers from some obvious difficulties as it relies on only one data point. In the Aged Averages (AVG) algorithm, a weighted average of the utilizations in the last few intervals is used as the prediction for the next interval [14]. Using more than one interval makes the algorithm more robust against transient changes in load. This is illustrated in Fig. 5.9. Here, the utilizations at intervals t and $t - 1$ are averaged with equal weights to generate the predicted utilization for interval $t + 1$. In Fig. 5.9(a), the utilizations for intervals t and $t - 1$ are 70% and 80% respectively, giving the predicted utilization for interval $t + 1$ to be 75%. The voltage is stepped down. In Fig. 5.9(b), intervals t and $t - 1$ have utilizations 90% and 70% respectively, giving 80% as the prediction for interval $t + 1$. This leads to no change in voltage levels, treating the 90% value as a transient when it appears for the first time. If the rate is sustained (for another interval in this case), then the aged average reflects the higher load and the voltage is eventually stepped up.

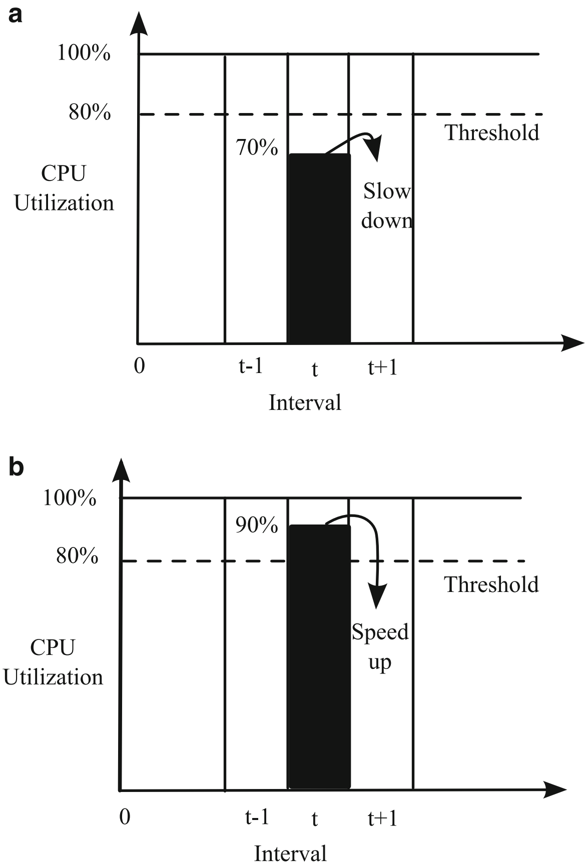


Fig. 5.8 *Past* Algorithm: predict the utilization in the next interval to equal that of the last interval (**a**) 70% utilization predicted for interval t+1. Slow down. (**b**) 90% utilization predicted for interval t+1. Speed up

To evaluate the effectiveness of the above algorithms, one can compare them with an *Oracle* algorithm that has advance knowledge of the next interval's load. Schemes such as AVG lie somewhere between the effectiveness of Past and Oracle, with the increasing effectiveness coming with the associated cost penalty of larger storage, which can be an issue in a hardware implementation. Variations of this strategy can be thought of in slightly different contexts, particularly ones involving the choice between different *power modes*: *active*, *sleep*, and *power down*. In general, we would like to move the system to power down mode upon encountering long idle periods so as to save power, but the associated penalty is that it takes a relatively large number of cycles to bring the system back to active mode. Being over-aggressive in powering down the system means high performance overheads incurred in waiting for the system to be usable again.

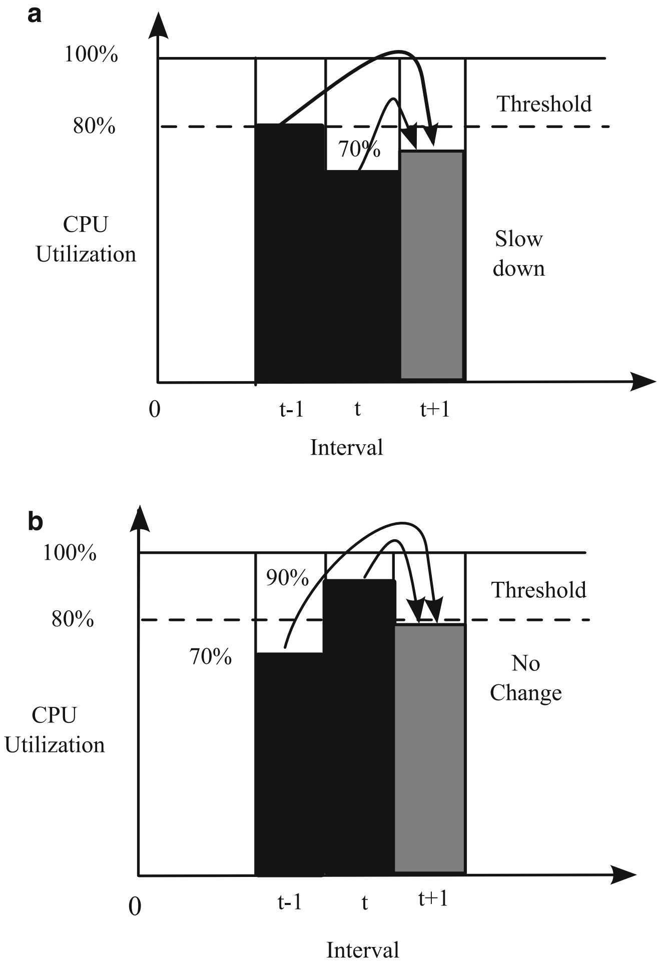


Fig. 5.9 *AVG* Algorithm: predict the utilization in the next interval to be weighted average of a few previous intervals (**a**) $(70 + 80)/2 = 75\%$ utilization predicted for interval t+1. Slow down. (**b**) $(70 + 90)/2 = 80\%$ utilization predicted for interval t+1. No change

5.1.2.1 DVFS in Real-time OS

Essentially, DVFS schemes use a simple feedback mechanism, such as detecting the amount of idle time on the processor over a period of time, and then adjust the frequency and voltage to just handle the computational load. This strategy has a simple implementation and follows the load characteristics closely, but cannot provide any timeliness guarantees and tasks may miss their execution deadlines. As an example, in an embedded camcorder controller, suppose there is a program that must react to a change in a sensor reading within a 5 ms deadline, and that it requires up to 3 ms of computation time with the processor running at the maximum operating frequency. With a DVS algorithm that reacts only to average throughput, if the total load on the system is low, the processor would be set to operate at a low frequency, say half of the maximum, and the task, now requiring 6 ms of processor time, cannot meet its 5 ms deadline. To provide real-time guarantees, DVS must consider deadlines and periodicity of real-time tasks, requiring integration with the real-time scheduler.

Let us look at some idealized situations in *real-time systems* to understand the limits of the applicability of DVFS.

First, let us consider a situation where we have tasks $T_1, \ldots, T_n$ to be scheduled in the time interval $[0, M]$. Each task has associated with it the number of required processor cycles R_i, the arrival time $A_i \in [0, M]$, and deadline for completion $D_i \in [0, M]$. The *voltage scheduling* problem is to find the optimal speeds at which the processor should work at every time instant in $[0, M]$ so that the total energy is minimized. It is assumed that the processor speed, and consequently the voltage, can be varied continuously, and can take all real values. The R_i values are fixed constants.

An optimal voltage scheduling algorithm uses the following greedy strategy [46]. For every time interval $I = [t_1, t_2]$ in the range $[0, T]$, find the *intensity* $g(I)$ defined as:

$$g(I) = \frac{\sum_i R_i}{t_2 - t_1} \tag{5.1}$$

for all i such that $[A_i, D_i] \in [t_1, t_2]$, that is, the intensity for interval I is computed considering all tasks whose arrival and completion times lie within the interval. Since $\sum_i R_i$ represents the total work that needs to be completed in time interval $I, g(I)$ represents the minimum required average speed of the processor during time interval I. Thus, if the processor is run at speed $g(I)$ during time interval I, it will be energy-optimal for this interval (if the speed is lower, then the tasks cannot complete; if the speed is higher, then the voltage – and hence energy – must be higher). We have established the speed/voltage values for interval $I = [t_1, t_2]$. Now, we just delete the interval from consideration, and recursively solve the same problem for the smaller interval thus obtained. The arrival and completion times of the remaining tasks are adjusted to reflect the deleted interval. This strategy gives the optimal speed/voltage assignment for minimizing energy [46].

The algorithm is illustrated in Fig. 5.10, with 3 tasks T1, T2, and T3, with the arrival times and deadlines being [0,5], [2,15], and [2,25] respectively, and number of cycles R_1, R_2, and R_3 being 1, 4, and 2 respectively. The intensities for the intervals are as indicated in Fig. 5.10(a). For example, the interval $I = [0, 15]$ has two tasks T1 and T2 with arrival/completion times lying within the interval, so $g(I) = \frac{R_1 + R_2}{15-0} = (1 + 4)/15 = 0.33$. Intervals not included in the list are those that cannot accommodate a single task. We select [0,15] for speed/voltage assignment first since this interval has the highest intensity. The assigned speed is 0.33. We then delete this interval, leading to a smaller problem indicated in Fig. 5.10(b). Only T3 still remains to be executed, and the arrival/completion times are as indicated in Fig. 5.10(b). Only one interval exists with intensity $\frac{R_3}{10-0} = 2/10 = 0.2$, and it

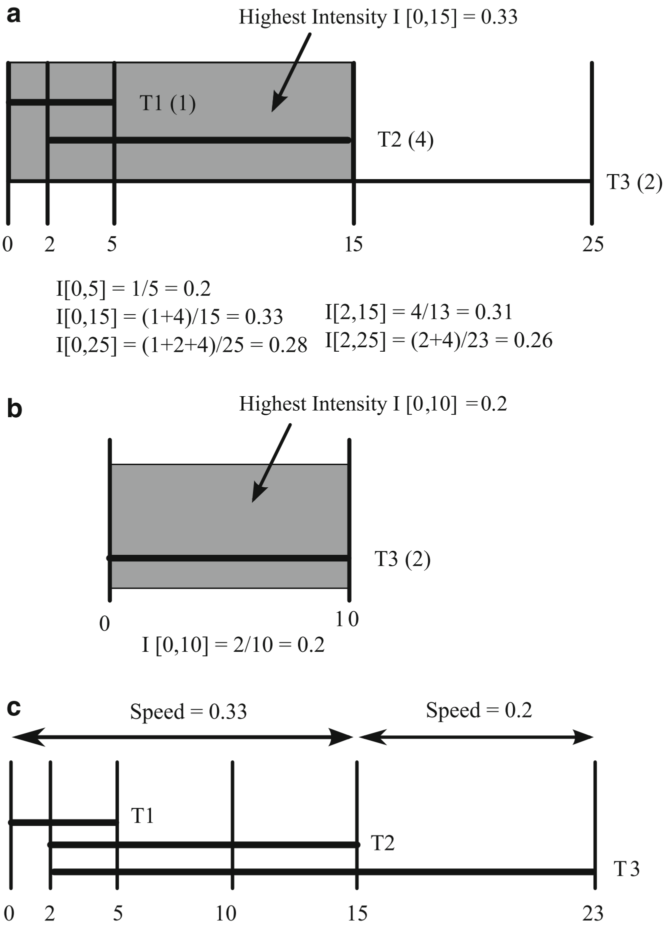

a
Highest Intensity I [0,15] = 0.33
T1 (1)
T2 (4)
T3 (2)
0
2
5
15
25
I[0,5] = 1/5 = 0.2
I[0,15] = (1+4)/15 = 0.33
I[2,15] = 4/13 = 0.31
I[0,25] = (1+2+4)/25 = 0.28
I[2,25] = (2+4)/23 = 0.26
b
Highest Intensity I [0,10] =0.2
T3 (2)
0
10
I [0,10] = 2/10 = 0.2
c
Speed = 0.33
Speed = 0.2
T1
T2
T3
0
2
5
10
15
23

Fig. 5.10 Optimal voltage scheduling. (**a**) Interval [0,15] has the highest intensity, so it is selected first, and speed 0.33 is assigned to it. The interval is then deleted. (**b**) Interval [0,10] (corresponding to the original interval [15,25]) is selected next, with speed 0.2. (**c**) The optimal voltage schedule corresponds to the two speed settings inferred in (**a**) and (**b**)

is trivially assigned speed 0.2. The speed assignment for the complete duration is summarized in Fig. 5.10(c). The processor runs at speed 0.33 for the first 15 time units, and 0.2 for the next 10 units.

The above problem formulation assumed that it is possible to change a processor's voltage and speed to *any* desirable value. In practice, we typically have to select from a set of discrete voltage settings for a processor. Let us address the problem of selecting the optimal voltages for running a processor, given a fixed load and a time constraint [19].

Our first observation, illustrated in Fig. 5.11, is that it is always sub-optimal to complete earlier than the specified deadline. Figure 5.11(a) shows two schedules,

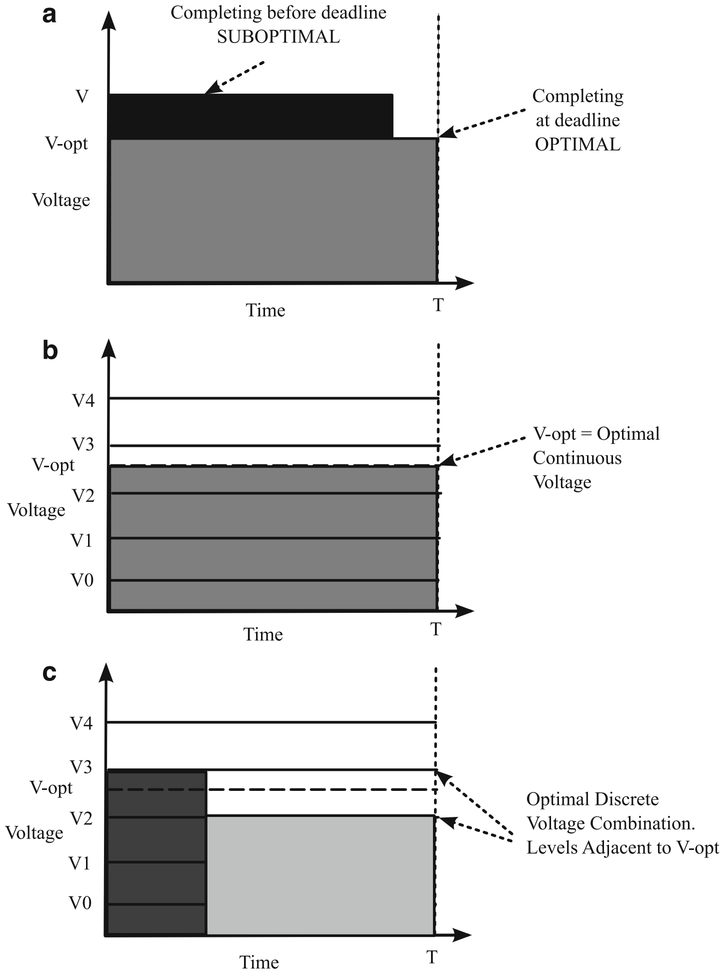


Fig. 5.11 Optimal discrete voltage scheduling with deadline T. (**a**) Energy is minimum when we select a voltage (V-opt) that allows us to complete the task exactly at the deadline. Any other voltage V leading to earlier completion is sub-optimal. (**b**) Discrete voltages available are: V0, V1, V2, V3, and V4. V-opt is not one of the available choices. (**c**) V2 and V3 are the two discrete voltage levels adjacent to V-opt. Energy is optimal when we use a combination of V2 and V3. Using any other voltage is sub-optimal

one completing before the deadline, followed by an idle period (during which the system could be assumed to dissipate zero power) with the voltage set at V; and the other completing at the deadline T. The task completing at T can progress at a lower voltage, which also increases the latency. However, the latency decreases linearly with the voltage whereas energy decreases as square of voltage. Hence, total energy is lower for the schedule completing at the deadline. In other words, average power (total latency divided by latency) is minimized for the task that utilizes all the time available. Let the corresponding voltage be V_{opt}. This result also follows from the application of the optimal algorithm discussed earlier. However, in reality, the voltage cannot be continuously varied, and we have to select from a set of discrete choices. The situation is illustrated in Fig. 5.11(b), where the permissible discrete voltages are: V_0, V_1, V_2, V_3, and V_4. We notice that V_{opt} is not one of the available voltages, so the optimal voltage/speed setting algorithm cannot be directly applied. It can be proved that in the discrete voltage scenario the optimal voltage for the processor will be a combination of the two discrete voltages adjacent to the computed optimal voltage V_{opt} [19]. From Fig. 5.11(b), we notice that V_{opt} lies between V_2 and V_3. As shown in Fig. 5.11(c), the energy-optimal solution is to run the system at voltage V_2 for some time, and at V_3 for the remaining time. The exact durations can be easily computed. Naturally, the resulting energy will be larger than the energy of running it at the hypothetical voltage V_{opt}, but the solution is still the best possible in the discrete voltage scenario. There is no need to consider other voltages. This is true even when voltage transitions are not immediate, as assumed in this discussion, but require a fixed duration [25].

The above conceptual treatment of the real-time DVFS problem made certain idealizations and simplifications that we need to be aware of, and also, exploit appropriately in a practical aggressive DVFS strategy. First of all, the number of cycles or any other measure of *work done* in a task may not be easily computed. This may be data dependent. Worst case execution times (WCET) need to be used. Of course, there may be many situations in which the worst case execution path is not exercised. Further, the presence of a memory hierarchy makes the WCET computation very difficult, and a theoretically guaranteed WCET that takes multiple levels of cache and secondary memory in its computation may be too pessimistic to be useful for DVFS.

Secondly, power of a large processor based system is not necessarily a quadratic function of the supply voltage, and latency is not necessarily inversely proportional to the supply voltage. These results hold for a single transistor, but there are other effects in a complex circuit such as that due to memory and I/O. Since DVFS modifies the delays of only the processor and does not touch components such as memory and I/O, the latter might actually consume more energy because they are held in active state for longer. Thus, practical DVFS strategies are more based on empirical models and prediction than theoretical analysis [44].

An example DVS-based optimization is shown in Fig. 5.12. *A* and *B* represent two code sequences following a memory load. *A* does not depend on the result of

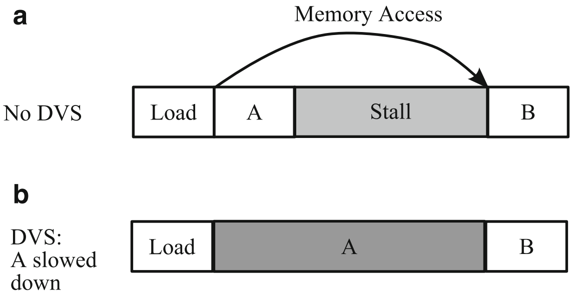


Fig. 5.12 DVS on stall due to memory access. (**a**) Lengthy stall due to load. *A* is not dependent on the load, so can proceed during the stall, but *B* cannot begin until the stall is resolved. (**b**) Anticipating the stall, we can slow down *A*, saving power

the load, but B does depend on it. In the situation shown in Fig. 5.12(a), A completes before the memory system has responded to the load request, leading to a system stall until the data is obtained, at which time B resumes execution. A possible resolution of this is shown in Fig. 5.12(b), where the system is aware of the expected latencies of A and the memory access. A can be slowed down by DVFS strategies to extend its execution time to be close to when the memory access is expected to complete. B resumes at its regular time, but the solution is more energy-optimal because A was executed at a lower voltage [44]. This optimization relies on reasonable estimates of the execution latencies being available, and the DVFS mechanism being able to respond with voltage/frequency switches fast enough to be useful.

One additional factor to be considered during DVFS is the accounting for leakage power. When the duration of a task is extended due to the voltage being scaled down, the dynamic energy decreases, but the leakage energy increases because the system continues to leak energy for the entire duration that it is active. Below a certain voltage/speed, the total energy actually increases [20]. The situation is shown in Fig. 5.13. The total system energy E consists of three components: (i) dynamic energy (E_d); (ii) leakage energy (E_l); and (iii) the intrinsic energy (E_{on}) that is necessary just to keep the system running. E_{on} consists of the power dissipated by system components such as analog blocks (phase locked loop and I/O) that are necessary for proper system operation.

$$E = E_d + E_l + E_{on}$$

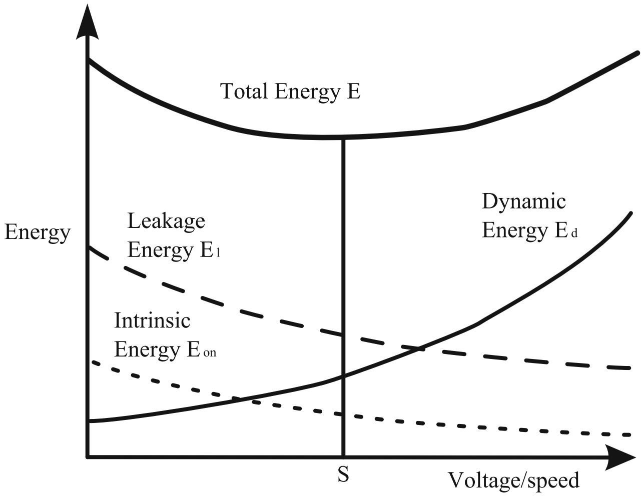


Fig. 5.13 Total system energy increases below critical voltage/speed S. At higher speed and voltage, dynamic energy dominates. At very low speed and voltage, the delay increases, so the system remains on for longer, leading to higher leakage energy

For a given time duration available for running a task, both E_l and E_{on} increase with decreasing voltage/speed; on the other hand, E_d decreases with decreasing voltage. The graph follows a 'U' shape, demonstrating a specific system-determined voltage/speed S that can be considered critical; below this level, the total energy starts increasing.

The DVFS concept is useful in the context of real-time systems where deadlines are imposed on tasks. Let us understand a straightforward application of *intra-task* DVFS – processor speeds/voltages are varied within an application so as to minimize energy. As seen earlier, the energy-optimal choice of voltage/speed is the one that causes the task to finish exactly at the deadline. However, different paths of a program will require different latencies depending on the amount of computation in the different branches, and also, as discussed earlier, the input data. A program is characterized by its worst case execution time (WCET), which, though hard to compute in general, could be obtained from user inputs on loop iteration counts, etc. Since the program path leading to the WCET must be executed within the deadline, the processor speed/voltage could be chosen such that this path completes exactly on the deadline.

Figure 5.14(a) shows a control flow graph (CFG) with each node corresponding to a basic block of code, and annotated by the delay in number of cycles required to execute it. Figure 5.14(b) shows a voltage/speed selection such that the worst case execution path A-B-D completes by the 80s deadline. However, there will be situations where this path is not executed, and the A-C-D path is taken. If the system is executed with the same voltage as in the A-B-D path, then the execution finishes by 40s, as shown in Fig. 5.14(c), and the system is idle for the remaining 40s. Instead, DVFS can be applied as soon as C starts executing, since the discrepancy between the remaining worst case execution time (10s for C + 20s for D = 30s) at the current voltage/speed, and the permitted time (70s) is known here. We can thus run C and D at a lower voltage/speed, thereby saving on the total energy, and yet meeting the task deadline Fig. 5.14(d). Although the actual decision is taken at run time, appropriate voltage scaling instructions can be inserted by the compiler in the C-branch [37].

In a *periodic* real-time system, we have a set of tasks to be executed periodically. Each task, T_i, has an associated period, P_i, and a worst-case computation time, C_i. Each task needs to complete its execution by its deadline, typically defined as the end of the period, i.e., by the next release of the task. The scheduling problem in this context is to assign the actual start times for all the tasks in such a way that all tasks meet their deadlines. Two important classical scheduling algorithms are noteworthy:

- **Earliest Deadline First (EDF).** In this strategy, we give the highest scheduling priority to the task that is constrained to complete the earliest.
- **Rate Monotonic Scheduling (RMS).** In this strategy, we give the highest scheduling priority to the task with the shortest duration.

While the EDF algorithm gives optimal results in terms of finding a valid schedule, the RMS is generally considered more practical for implementation.

The real time scheduling algorithms have to be appropriately adapted in order to accommodate DVFS possibilities. In addition to the traditional *schedulability*

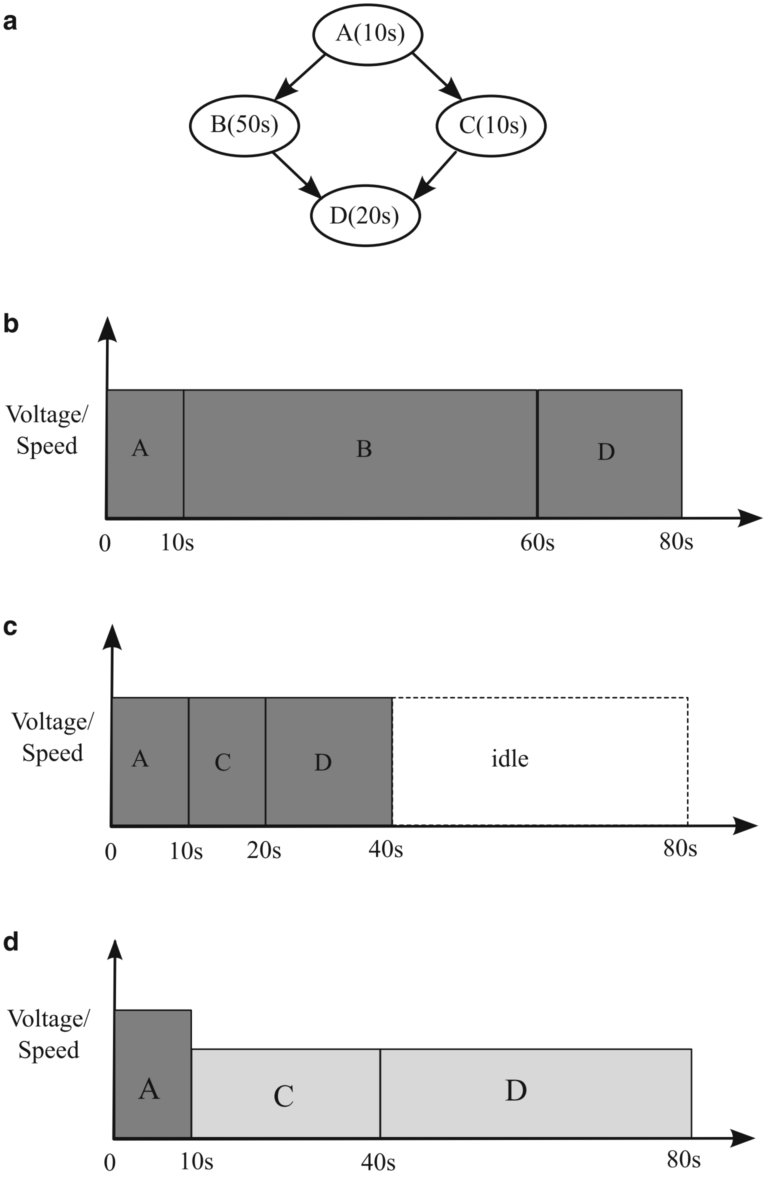


Fig. 5.14 Intra-task DVFS by lower voltage operation on slower path. (**a**) Control Flow Graph with node execution times. (**b**) Path A-B-D requires WCET = 80. (**c**) Path A-C-D completes faster. The system remains idle for 40s. No DVFS. (**d**) DVFS on the A-C-D path leads to slower execution of C and D, leading to energy saving

metric, an additional optimization criterion is total energy dissipation. For example, the necessary and sufficient schedulability test for a task set under ideal EDF scheduling requires that the sum of the worst-case utilizations be less than one, i.e.,

$$C_1/P_1 \ + \ ... \ + C_n/P_n \leq 1$$

When we apply DVFS, the operating frequency can be scaled by a factor $\alpha(0 < \alpha < 1)$, which in turn implies the worst case computation time of a task is scaled by a factor $1/\alpha$. The EDF schedulability test with frequency scaling factor α will then be:

$$C_1/P_1 \ + ... \ + C_n/P_n \leq \alpha$$

Operating frequency can then be selected as the least frequency at which the schedulability criterion is satisfied. The minimum voltage that will allow the system to operate at the required frequency is then chosen as a consequence. As shown in Fig. 5.15, this solution finds one constant operating point; the frequency and voltage do not change with time.

If each task T_i actually requires its worst-case time C_i to execute, then this result is optimal. However, in reality a task may often finish much faster than its worst

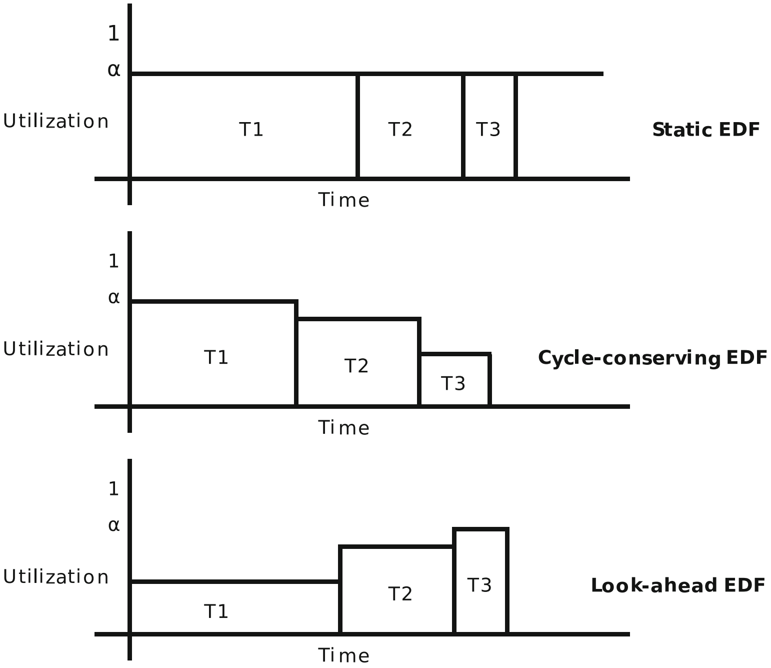


Fig. 5.15 Static EDF finds one fixed operating point at which the system must be executed to minimize power. Cycle-conserving and look-ahead schemes can change the operating point after each task. Look-ahead technique is able to exploit the difference between the actual execution time and the worst case execution time of the task

case time. Thus if c_i is the actual execution time of task T_i, and $c_i < C_i$, then this extra slack may be utilized to further slow down the system and save power. If c_i is used to compute the utilization instead of C_i, then the operating frequency obtained will result in a task set that is schedulable, at least until T_i's deadline. This is because the number of processing cycles available to other tasks between T_i's release and deadline is unchanged from the case where WCET is assumed and is actually needed. When $c_i \ll C_i$, this scheme, called *Cycle-conserving EDF* can exploit the excess slack to optimize power. Cycle-conserving EDF assumes the worst case initially, executes at a high frequency until some tasks complete, and only then reduce operating frequency and voltage. In contrast, the a look-ahead approach could defer as much work as possible, and set the operating frequency to meet the minimum work that must be done now to ensure all future deadlines are met. While this implies that high operating frequencies may be needed later on, but again, if $c_i \ll C_i$, this scheme will be advantageous. Experimental results show that the look-ahead approach is the most effective, but both the approaches can significantly reduce the energy consumption by exploiting the slack between the C_i, and c_i of the tasks.

5.1.3 I/O Device Power Management

I/O devices on computer systems such as disks and monitors dissipate a significant amount of power, and modern operating systems support various features for power management of these devices. The simplest strategy here is to monitor the activity pattern on these devices, and when the idle duration exceeds a certain threshold, move them into a low power state. For example, a laptop computer usually offers a user-configurable setting for the idle duration after which the LCD display is turned off; this saves a considerable amount of power. More sophisticated techniques in this line include activating a laptop camera periodically to monitor the surroundings; if a human face is not detected, the display could be turned off.

Simple power management mechanisms are also applicable to the hard disk. Since hard disks consume significantly lower power when in sleep mode, an idle-duration based decision to move the hard disk to sleep state is appropriate. Naturally, prediction mechanisms come handy for making the critical decision of how long we should wait before spinning a disk down. Doing it late implies wasted power but better performance. Spinning down early leads to poor performance if too many restarts are necessary, along with higher power because restart may be expensive in terms of power. Moreover, disks need not be completely spun down. Disk power management can also be performed at a finer level, where we can gradually vary the rotation speed of the disks [15].

Certain non-trivial implications of disk power management decisions ought to be kept in mind. Moving disks to sleep state implies that dirty buffers are written less often to the disk – once in several minutes instead of a few seconds – to enable the disk to stay longer in low power mode. This increases the possibility of data loss due to power outages, where the disk does not get an opportunity to synchronize with the modified buffers. Further, frequent spin up and spin down of disks causes reliability problems and may lead to early failures. Overall, secondary storage in computer systems is a rapidly evolving area; hard disks face stiff competition from other technologies such as non-volatile memory as the latter has advantages with respect to power, weight, and noise, and is getting close with respect to cost and density.

5.2 Compiler Optimizations

Compiler optimizations targeting high performance generally also reduce average power and energy indirectly. When the optimized code generated by a compiler results in lesser number of instructions executed, it also means a smaller number of accesses to instruction memory. Since energy consumed by memory is proportional to the number of accesses, this also reduces the total energy consumed. Along the same lines, optimizations that reduce the number of accesses to data memory also reduce the total memory energy consumption. Thus, for example, all register allocation related optimizations, which increase the efficiency of register usage, are also favorable with respect to power, as it is more power-efficient to access registers than memory. This argument also generalizes to other levels of the memory hierarchy. Performance optimizations that increase the hit ratio to the L1 cache are also indirectly energy optimizations, since the L1 cache access dissipates lesser energy than an L2 cache access. The extent of performance improvement due to a compiler optimization may be different from the extent of power improvement. However, the optimizations are generally in the same direction, and if a memory related optimization improves performance, then it also reduces power and energy. However, interesting exceptions do exist – good examples being those that rely on speculative memory loads. In such cases, the access latency may be hidden by other CPU activity, but the associated energy dissipated cannot be undone. Such an optimization improves performance, but reduces energy efficiency.

Making the compiler explicitly aware of the performance/energy optimized features present in the memory subsystem increases the compilation time, but yields the power benefits without any run-time overhead and without the need of expensive hardware. While most standard compiler optimizations including constant folding and propagation, algebraic simplifications, copy propagation, common subexpression elimination, loop invariant code motion, loop transformations such as pipelining and interchange, etc. [32], are also relevant for power reduction, some others that increase the code size (such as loop unrolling and function inlining) need more careful attention. Optimizations such as unrolling and inlining increase the code size, thereby increasing the instruction memory size. Since larger memories are associated with increased access energy, these transformations may actually end up decreasing energy-efficiency.

5.2.1 Loop Transformations

Loop transformations such as *loop interchange, loop fusion, loop unrolling*, and loop tiling, which typically result in better cache performance through exploiting data reuse, also lead to improvements in power/energy by way of minimizing accesses to off-chip memory. Transformations such as unrolling cannot be indiscriminately applied because they lead to cache pollution, which affects performance; the same argument also applies to power, as we usually use cache misses as the evaluation metric.

However, other transformations such as *scalar expansion* work in the opposite direction. In scalar expansion, a global scalar variable shared across iterations that prevents parallelization, is converted into an array variable to remove the data dependency and parallelize the independent iterations. As the new array is mapped to memory (instead of possibly a register earlier), such an optimization results in a larger number of memory accesses and the associated address calculation, and consequently, worse power [23].

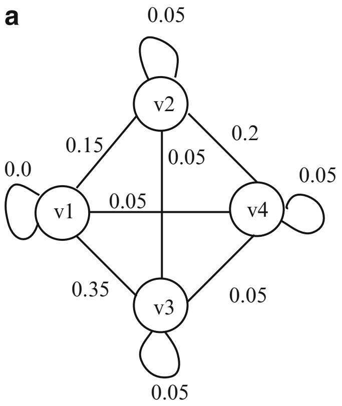
5.2.2 Instruction Encoding

When a new instruction is fetched into the instruction register (IR), several bits of the current IR are switched. The switching activity during the instruction fetch phase is directly proportional to the number of bits switched in the IR between the successively fetched instructions. The bit changes on the opcode field can be decreased by assigning opcodes so that frequently occurring consecutive instruction pairs have a smaller Hamming Distance between their opcodes.

We can represent the instruction transition frequencies as an instruction transition graph (ITG) $G = (V, E, w)$ where V is a set of instructions, E is the set of undirected edges between all the elements in V, and w is a probability density function that maps each edge $e = (v_1, v_2) \in E$ to a real number between 0 and 1. $w(e)$ indicates the relative frequency of the instruction transitions between v_1 and v_2.

Given an instruction transition graph G, a set S of binary strings of length $\lceil log_2|V|\rceil$, and an opcode assignment function $f : V \rightarrow S$, a power metric, the average switching in G under f can be defined as $P = \sum w(e) \times h(f(v_1), f(v_2))$, where h is a function returning the Hamming Distance between two binary strings. This is illustrated in the example shown in Fig. 5.16(b). Figure 5.16(a) shows the graph with nodes v1 to v4, each representing an instruction. The edges connecting these nodes are annotated with the instruction transition probabilities. Figure 5.16(b) shows the encoding of these instructions and Fig. 5.16(c) shows the cost incurred due to the transitions shown in Fig. 5.16(a) when the encoding shown in Fig. 5.16(b) is used.

For low-power opcode encoding, the goal is to find an optimal opcode assignment function f_{opt} that minimizes the power consumption. Standard finite state



b

Inst.	Opcode
v1	00
v2	01
v3	10
v4	11

c

Inst. Seq.	Hamming Dist.	Cost
v1 ↔ v2	1	1 x 0.15
v1 ↔ v3	1	1 x 0.20
v1 ↔ v4	2	2 x 0.05
v2 ↔ v3	2	2 x 0.05
v2 ↔ v4	1	1 x 0.35
v3 ↔ v4	1	1 x 0.05
v1 ↔ v1	0	0 x 0.05
v2 ↔ v2	0	0 x 0.00
v3 ↔ v3	0	0 x 0.05
v4 ↔ v4	0	0 x 0.05
Total Cost		0.95

Fig. 5.16 An example showing the computation of the cost associated with an encoding of instructions when applied to a given Instruction Transition Graph. (**a**) Example instruction Transition Graph. (**b**) Instruction encoding. (**c**) Computation of cost

machine encoding techniques can be adapted for this purpose. Further reduction in IR switching can be effected by changing the register numbers in a generated binary to minimize the switching in register numbers in consecutive instructions.

5.2.3 Instruction Scheduling

In the instruction encoding discussed above, we assume that the instruction sequence is fixed. Instruction scheduling is the complementary optimization, where we exercise the flexibility to re-order instructions to minimize bit switching. Here, sequences of instructions can be re-scheduled where permissible to reduce transition count on the instruction register and the instruction memory data bus. Additionally, we can re-label registers in the generated instructions such that bit switching in consecutive instructions is reduced [31,33,39,41].

In VLIW processors, different instructions may have varying number of operations, leading to a significant variation in the *step power* (difference in power between consecutive clock cycles) and *peak power* (maximum power dissipation during program execution). Both step power (which affects inductive noise) and peak power affect system reliability. A more balanced distribution of instructions in the schedule that avoids the extremes in terms of number of instructions in a cycle and transitions between them leads to better step power and peak power behavior. Since the instruction stream in VLIW processors is usually compressed, a reordering of the instructions within the same long word may lead to a better compression. The compression implications of different orderings can be evaluated by the compiler and the best one generated, ultimately leading to fewer I-Cache misses. Keeping in view the transition activity on the instruction bus, the instructions within the same VLIW instruction word can be re-ordered to minimize the Hamming distance from the previous instruction word. This can also be done across words, if the performance is not affected [6,26,36,47].

Compared to the run-time environment, a compiler has a deeper view of the individual application being compiled, and can perform optimizations spanning a large section of code. In a hybrid VLIW/Superscalar architecture, a low-power enhancement to a superscalar processor is used, where, if the compiler is able to find efficient instruction schedules, then the low power mode is used and the circuitry for dynamic scheduling is turned off [43].

5.2.4 Dual Instruction Set Architectures

The Instruction Set Architecture (ISA) forms the interface between the hardware and software, and it is the compiler's task to convert an application expressed in high level language in terms of machine instructions. The instruction set itself has a very significant impact on the power-efficiency of program execution.

Traditionally, ISAs have been of fixed width (e.g., 32-bit SPARC, 64-bit Alpha) or variable width (e.g., x86). Fixed width ISAs give good performance at the cost of code size and variable width ISAs give good performance at the cost of added decode complexity. However, neither of the above are good choices for low power embedded processors where performance, code size, and power are critical con-

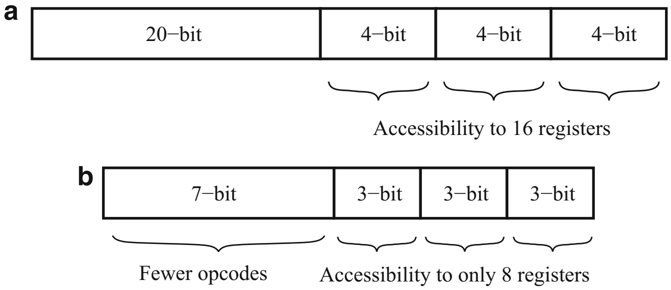

a
20–bit
4–bit
4–bit
4–bit
Accessibility to 16 registers
b
7–bit
3–bit
3–bit
3–bit
Fewer opcodes
Accessibility to only 8 registers

Fig. 5.17 Reduced bit-width Instruction Set Architecture or rISA is constrained due to bit-width considerations. Consequently, rISA instructions often have access to only a fraction of the register file. (**a**) 32-bit normal instruction. (**b**) 16-bit rISA instruction

straints. Dual width ISAs are a good trade-off between code size flexibility and performance, making them a good choice for embedded processors. Processors with dual width ISAs are capable of executing two different instruction sets. One is the "normal" set, which is the original instruction set, and the other is the "reduced bit-width" instruction set that encodes the most commonly used instructions using fewer bits (Fig. 5.17).

A good example of a dual-width ISA is the ARM [1] ISA with a 32-bit "normal" Instruction Set and a 16-bit Instruction Set called "Thumb". Other processors with a similar feature include the MIPS 32/16 bit TinyRISC [29], ST100 [38], and the Tangent A5 [3]. This feature is called the "**r**educed bit-width **I**nstruction **S**et **A**rchitecture" (**rISA**).

Processors with rISA feature dynamically expand (or translate) the narrow rISA instructions into corresponding normal instructions. This translation usually occurs before or during the decode stage (Fig. 5.18). Typically, each rISA instruction has an equivalent instruction in the normal instruction set. This makes translation simple and can usually be done with minimal performance penalty. As the translation engine converts rISA instructions into normal instructions, no other hardware is needed to execute rISA instructions. If the whole program can be expressed in terms of rISA instructions, then up to 50% code size reduction may be achieved. Code size reduction also implies a reduction in the number of fetch requests to the instruction memory. This results in a decrease in power and energy consumption by the instruction memory subsystem. Thus, the main advantage of rISA lies in achieving low code size and low energy consumption with minimal hardware alterations. However, compiling for rISA instructions is complicated due to several reasons:

- **Limited Instruction Set:** The rISA instruction set is tightly constrained by the instruction width. Since only 16 bits are available to encode the opcode field and the three operand fields, the rISA can encode only a small number of normal instructions. Therefore several instructions cannot be directly translated into rISA instructions.

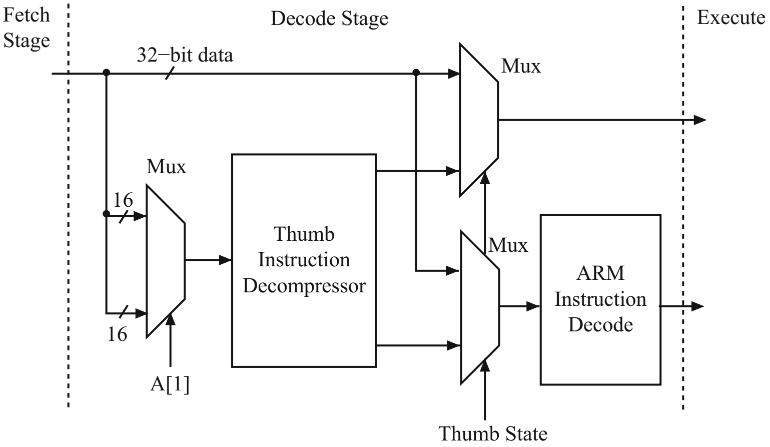


Fig. 5.18 rISA instructions are translated to normal instructions before or during decode. This allows the rest of the processor to stay unchanged

- **Access to only a fraction of registers:** The rISA instruction set, because of bit-width restrictions, encodes each operand (such as register address) using fewer number of bits. Therefore, rISA instructions can access only a small subset of registers. For example, the ARM Thumb allows access to 8 registers out of the 16 general-purpose ARM registers.
- **Limited width of immediate operands:** A severe limitation of rISA instructions is the inability to incorporate large immediate values. For example, with only 3 bits available for operands, the maximum unsigned value that can be expressed is 7.

Because of the problems mentioned above, indiscriminate conversion of normal instructions to rISA instructions may actually increase code size and power consumption, not only because a normal instruction can map to multiple rISA instructions, especially if it has large immediate operand fields, but also because of spill code since rISA instructions can access only a limited set of registers.

One of the most important decisions in a rISA compiler is the granularity at which to perform the conversion. The conversion can be performed at routine level granularity, where all the instructions in a routine can be in exactly one mode – the normal mode or the rISA mode. A routine cannot have instructions from both ISAs. Routine-level rISAization (the process of conversion from normal instructions to rISA instructions) has some drawbacks:

- First, a routine-level granularity approach misses out on the opportunity to rISAize code sections inside a routine that is deemed non profitable to rISAize. It is possible that it is not profitable to rISAize a routine as a whole, but some parts of it can be profitably rISAized. For example, in Fig. 5.19(a), Function 1 and Function 3 are found to be non-profitable to rISAize as a whole. Routine-level granularity approaches will therefore not rISAize these routines.

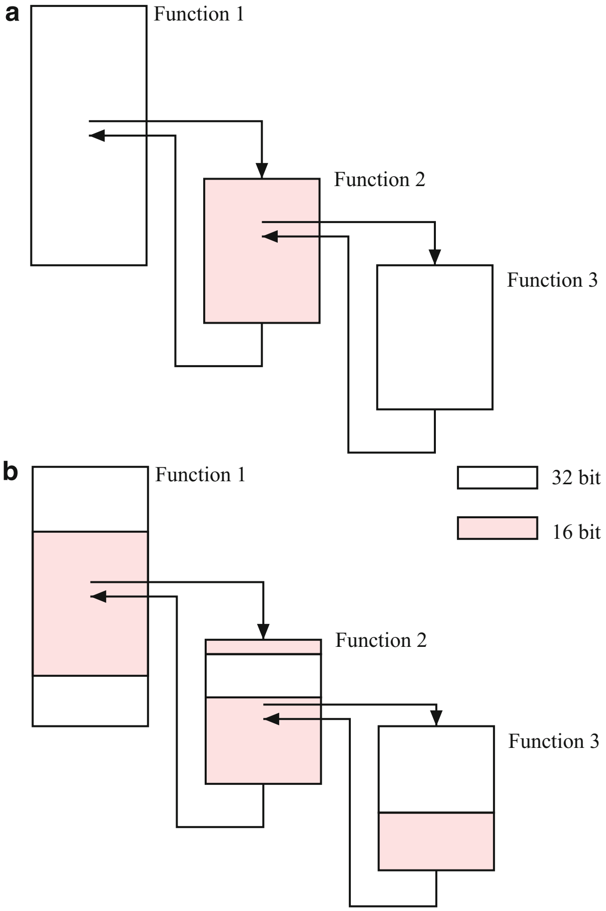


Fig. 5.19 rISAization at function level has very little overhead, but misses out on the possibility of selectively converting only the profitable regions of a function. (**a**) Routine Level Granularity. (**b**) Instruction Granularity

- Secondly, with routine-level rISAization, it is not possible to exclude from conversion some regions of code inside a routine that may incur several register spills. It is possible that excluding some pieces of code inside a profitable routine may increase the code compression achieved. For example, in Fig. 5.19(b) the instruction-level granularity approaches have the choice to exclude some regions of code inside a routine to achieve higher code compression.

Performing rISAization at instruction-level granularity alleviates both the above problems, as we can rISAize profitable portions of the application code, while excluding the non-profitable parts. However, rISAizing at instruction-level comes with its own set of challenges. Foremost is the overhead of the mode change operation: the instruction that informs the processor that the following instructions are in the normal mode, or the rISA mode. In processors that implement routine-level conversion, this functionality can be added to the function call instruction, but instruction-level conversion requires explicit instructions. The direct implication of this is that converting only a few instructions will not be profitable, and several contiguous instructions must be converted to overcome the conversion overhead and obtain code size and power improvements. Since basic blocks are typically small, a good approach requires an inter-basic block analysis for conversion. Further, an effective approach also necessitates an associated scheme to estimate the register pressure in a code segment in order to more reliably compute the increase in the code size by rISAizing the code segment.

Experimentation with the rISAization strategy shows that rISA is a very effective code size reduction, as well as power reduction technique, and a smart compiler can consistently achieve upwards of 30% reduction in code size, and similar reduction in the power consumption of the instruction cache.

5.2.5 Instruction Set Extension

Instruction set extension is the process of adding new instructions in the processor, and adding the corresponding functional unit and control circuitry to enable the detection and execution of the new instruction, with the objective of improving the power and performance of the processor. This is specially useful in application specific processors (ASIPs), where there may be some large pieces of functionality that are used very often, and the application could benefit from performing it directly in hardware. Consider a cryptographic application using elliptic curve encryption to encode data. A processor used for this application could greatly benefit if the entire elliptic curve encryption could be performed as one single instruction, rather than as a sequence of smaller instructions. One common application for instruction set extension is the MMX extension to the x86 architecture that provides special instructions for SIMD arithmetic and string manipulation.

The procedure for extending the instruction set of a processor starts with identifying commonly occurring instruction patterns in the application set of interest, replacing them by a new instruction in the application code, adding a new hardware unit to execute the new instruction, and finally adding control logic to decode, issue, and commit the instruction.

An Instruction Set Extension or ISE typically encapsulates multiple atomic operations constituting the critical portion of the application. Execution of an ISE on a custom unit effectively migrates multiple operations from software to hardware, thus greatly accelerating the application performance. Along with performance, there are other obvious benefits of such application-specific processor customization. Because of compacting multiple operations into a single ISE, there is an overall code size reduction. Furthermore, we can expect energy reduction because fewer instructions are executed for every replacement of a large set of operations by the ISE. Such replacement causes reduced switching activity due to reductions in the number of fetch, decode, and register store operations.

Automatic generation of ISEs is a key, and perhaps the most crucial step in automating the process of processor customization. To do this, the Control Flow Graph (CFG), and the Data Flow Graph (DFG) of the basic blocks of the application must be abstracted. DFG is a Directed Acyclic Graph (DAG) $G = (V, E)$, where the nodes V represents the instructions or external inputs/outputs and the edges E capture the data dependencies between the nodes. A cut $C \subseteq G$ can be a potential ISE if it satisfies some conditions:

- **Forbidden Operations:** Due to microarchitectural restrictions, operations of a certain type might not be allowed within the cut. For example, memory operations have been traditionally prohibited in the process of ISE generation. This is because, first of all, if memory operations are allowed in ISEs then the ISE must be combined with the load/store unit. Otherwise, the custom unit must have a new connection (that could be shared) to the memory, causing coherency issues between the data shared by the custom unit and the rest of the processor. Therefore, when searching for a cut, we have to find a maximal cut that does not contain any node that cannot be a part of the ISE.
- **Input-Output Constraints:** The custom unit will receive its operands from a register file (shown in Fig. 5.20). As a result, the number of source and destination operands of the new instruction is limited by the number of read and write ports respectively in the register file. For embedded processors with relatively fewer read/write ports on the Register File, this can be a crippling limitation.
- **Convexity Constraint:** Only convex cuts can be a candidate for ISE. In a convex cut C there exists no path from a node $u \in C$ to another node $v \in C$ through a node $w \notin C$. This is needed because scheduling policies in processors typically assume that all operands of an instruction are read before the instruction starts execution. Implementing a non-convex graph would require significant changes

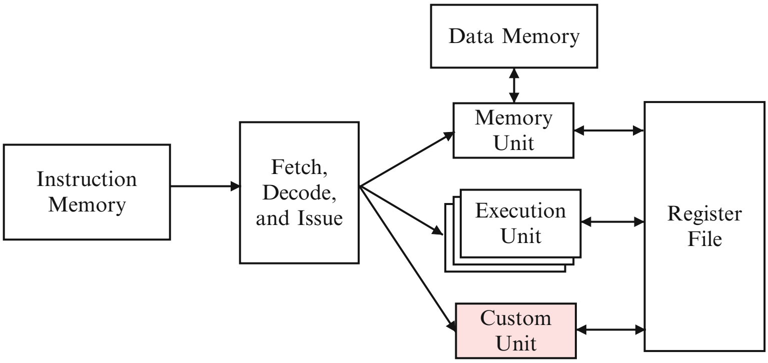


Fig. 5.20 In tightly-coupled processors, a custom unit is tightly integrated with the processor pipeline to implement instruction set extension functions

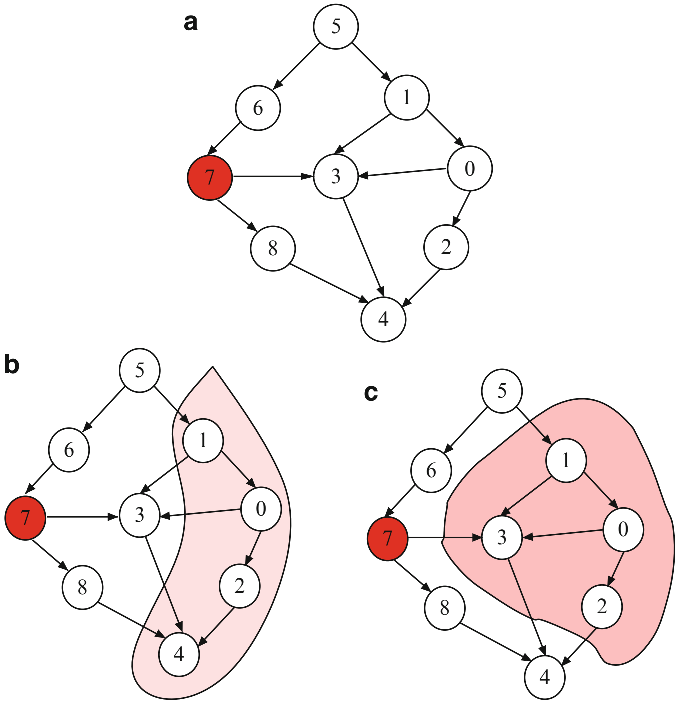


Fig. 5.21 The objective of ISE generation is to find a maximal cut that does not have forbidden functions (Shaded Nodes) and satisfies input-output and convexity constraints. (**a**) Graph with Max. Inputs = 3, Max. Outputs = 2. (**b**) Invalid Cut. (**c**) Valid Cut

in the scheduling policy. Figure 5.21(c) shows a valid cut, while Fig. 5.21(b) shows a cut that violates convexity constraints. This is because there is a path from node 1 to node 4 (both in the cut) that goes through node 3 (outside the cut).

Thus the problem of finding an ISE is to find non-overlapping cuts $C_i \subseteq G$ that satisfy the input-output, convexity, and forbidden operations constrains, and maximize the improvements in power and performance. This dual objective is tricky because on one hand, finding as large a cut as possible is beneficial, but on the other hand, the cut should be relatively small so that it is generic enough to have several instances in the application, to deliver good results. Integer Linear Programming (ILP) solutions have been developed, with the predictable behavior of generating optimal results but at the expense of too much time; they work well on small DFGs, and therefore are unable to find large cuts. On the other hand, heuristics have a hard time finding large cuts. Monte carlo and genetic solutions have also been explored. However it is difficult to define good fitness function and the termination criteria. Clustering techniques [11, 40] start with a seed node and use a guide function to select the best direction to grow the cluster. One technique prunes the candidates that do not reach a certain percentage of the best priority discovered so far, while the other prunes the directions of search that are not estimated to be worthy for growing a candidate. ISEGEN [5] uses a graph partitioning scheme based on the Kernighan-Lin heuristic. On multimedia benchmarks and a processor with 4 read ports and 2 write ports on the Register File, an average of 50% speedup and is reported. The power savings also fall in the same range.

5.2.6 Power Gating

The compiler has an intimate knowledge of the processor microarchitecture. This has been exploited to develop several compiler techniques to modify the application, so that it executes in a power-efficient manner on the microarchitecture. Among various techniques proposed for leakage energy reduction at the microarchitecture level, power gating has emerged as one of the most promising approaches [8, 34]. In this technique, leakage power is reduced by shutting off the power supply to the FU during periods of inactivity (Section 3.5.3) [21].

Figure 5.22 shows the estimated energy density of different components in the ALPHA DEC 21364 processor while executing a representative susan-corners benchmark from the MiBench suite on PTScalar [27] simulator. The ALUs have the second highest energy density among all the units, next only to the integer register file. This observation is also consistent with other studies such as [12], where it is reported that compared to large modules such as secondary caches, FUs are very

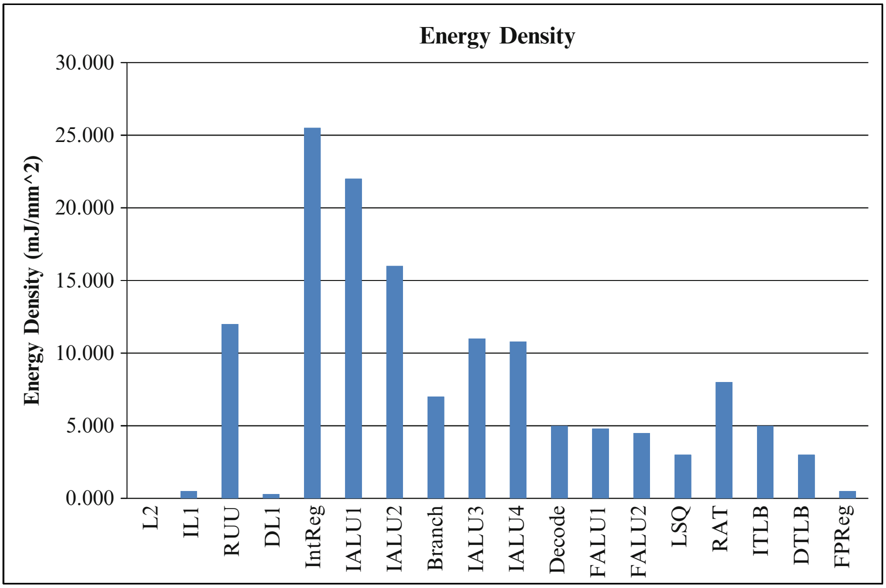


Fig. 5.22 Power gating of functional units is important, as they are typically the most important hotspots in the processor

active blocks with power densities up to twenty times higher. High power densities directly result in high temperature, which ultimately makes function units some of the highest leakage sites in the processor.

Power gating promises to be an effective approach for containing the leakage power of FUs. However, power gating large logic structures such as ALU require a large sleep transistor (see Section 2.5.5). Synthesis results at 65nm show that the delay of the sleep transistor will be about 6-10 processor cycles with a 3 GHz clock. Given this, the problem of power gating FUs translates to finding idle intervals of inactivity of the FUs, and power gating the FUs during these periods. The good news is that inherent instruction dependencies in programs ensures that we cannot use all FUs all the time. Hence, idle periods on FUs are a commonly occurring phenomenon.

One popular power gating technique is based on FU idle periods [42]. Here, the activity of FUs is monitored, and if an FU is idle for more than a threshold t_{idle} cycles, the power supply to the FU is gated off. The control circuit for power gating each FU is local and independent of other FUs. Once in a power-gated state, the FU will be woken up (power gating is disabled) when an operation is issued to it. Power gating has also been attempted in a VLIW compiler by issuing instructions to turn the FUs on or off. This is typically done at a loop-level – the number of FUs required for a loop is determined, and those not needed are turned-off. However, in order to not overheat the few active FUs, the activity is circulated among the FUs, turning them on and off in an iteration.

Use of tiny leakage sensors deployed on each FU can lead to further optimization of FU leakage power [24]. This approach attacks the power gating problem in two steps. First, it looks at the recent history of execution and determines how many FUs to keep "on". Second, it power gates the FU whose leakage is the least. Operations are issued only to the "on" FUs. Since the decision of which FUs to keep "on" is based on the leakage of the FU, it automatically considers the usage, temperature, and also the process variation effects. Because of process variations (manufacturing inaccuracies), FUs can have different base leakages. This is an exponentially growing problem as we tread towards finer dimensions in manufacturing. Leakage-aware power gating automatically considers this process variation effect, and is able to "even-out" the leakage of the FUs (Fig. 5.23).

5.2.7 Dynamic Translation and Recompilation

One traditional handicap of the compiler with respect to power optimization is that it has a limited view of the run-time environment. Since the compiler is unaware of what other tasks would be simultaneously contending for common system resources, it is difficult for it to be aggressive in its power optimizations. *Dynamic translation* and *dynamic recompilation* refer to techniques where a certain amount of code generation is actually performed by the CPU in hardware at execution time. The Transmeta Crusoe processor provided an early glimpse into such possibilities in a commercial setting [13]. A VLIW-style architecture was adopted with a view to

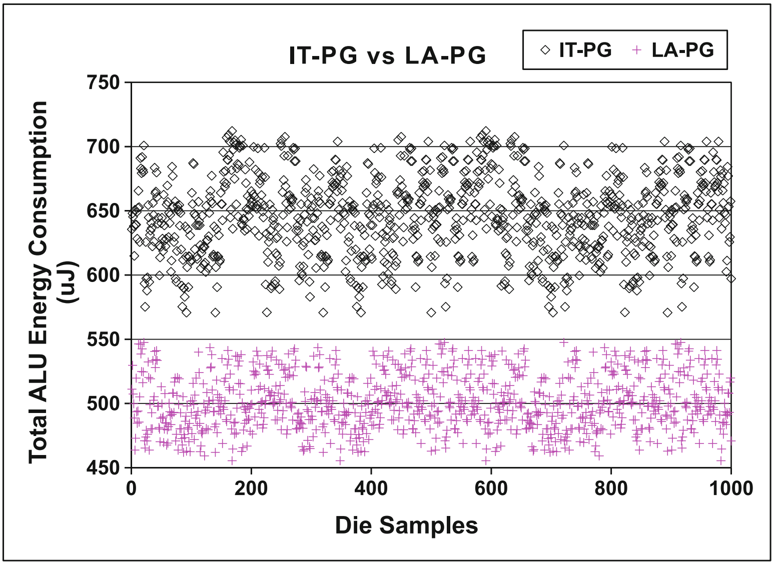


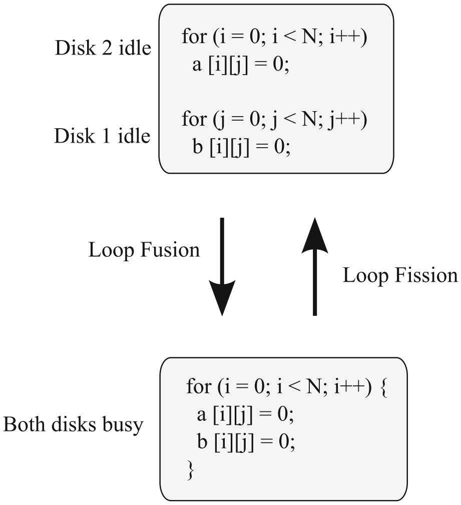
Fig. 5.23 Leakage-Aware power gating helps not only in reducing the leakage of FUs, but also helps in reducing the variation in the leakage due to process variations

reducing the power overhead of performing major tasks such as instruction reordering. Instead, a run-time software binary translator was used to generate the VLIW instructions from the original x86 code on the fly, a small sequence at a time. This still led to a significant overhead the first time the code was translated, but the resulting decoded VLIW code was cached so that future accesses to the same instruction could be read from the local memory, without the power overhead of decoding and instruction re-ordering.

The Crusoe processor was an early instance of a laptop class processor that could be run at several different voltage and frequency settings. In addition to the dynamic translation, a dynamic recompilation feature was also introduced, which would monitor the execution carefully to find frequently executed sections of code and generate optimized versions at run time. The dynamic translation and optimization feature has, since then, been implemented by several newer generation processors.

5.2.8 Compiler Optimizations Targeting Disks

Since accesses to the disk involves a significant amount of energy, making compiler optimizations disk-aware can help reduce overall system power. Both data layout and instruction transformations can benefit from knowledge of the disk subsystem. For example, data can be laid out in such a manner that in a parallel disk system, only a few disks are continuously accessed, generating the opportunity



```
for (i = 0; i < N; i++)
  a [i][j] = 0;

for (j = 0; j < N; j++)
  b [i][j] = 0;
```

```
for (i = 0; i < N; i++) {
  a [i][j] = 0;
  b [i][j] = 0;
}
```

Fig. 5.24 The loop fusion transformation could be bad for disk power when data from different disks are accessed in the merged loop. In this example, arrays *a* and *b* reside on Disks 1 and 2 respectively. When the loops are split (left), Disk 2 can be powered down during the first loop and Disk 1 can be powered down during the second loop. When the loops are fused, both disks are busy throughout the merged loop

to power down the remaining ones. In this context, it is worth re-examining the implications of typical compiler optimizations – some of them work in the opposite direction in this context. An interesting observation is that *loop fusion* can be detrimental from the point of view of disk power, especially when it leads to additional arrays being accessed – and hence, more disks being activated simultaneously [22]. The reverse optimization, *loop fission*, can be beneficial using the same argument (Fig. 5.24). Note that this is in contrast to the previous observation in Section 5.2.1.

5.3 Application Software

Power awareness at the level of the hardware, operating systems, and compilers, is gradually finding its way to application software through application programming interfaces (APIs) that expose the underlying power management facilities. These APIs can be used by the programmer to pass useful information on to the operating system – specific information about the application's behavior that might not be easy to infer automatically. The converse is equally useful – knowledge provided by the operating system helps the application tune itself to the state of the system resources.

5.3.1 Application-aided Power Management

One class of hints that can be provided by an application includes task completion deadlines, expected execution times, and other measures of the estimated complexity of the task that might not be easily available statically, but could be present or computed at run time. Such information can help the operating system make more informed power management decisions.

Exposing the state of different system resources to an application can help building systems that adapt themselves dynamically to achieve better power efficiency. An example is when there is a choice to fetch a piece of data from multiple sources – a disk and the network. If the current power modes (and associated performance penalties) of the connected devices were available to the application, a quick estimate could help decide the most appropriate device for servicing a request. If the disk is powered down, it may be cheaper in terms of both performance and energy to fetch relatively small-sized data from other networked devices [2, 44]. A general handling of this situation needs some additional intelligence. If a sequence of such small-sized requests are issued, then, beyond a certain count, it would be more energy-efficient to wake up the disk instead. A co-ordinated strategy is shown to be useful. Involving the application in the power management decision is useful here. The application, which may have knowledge about the future request patterns it will issue, can take the decision about the optimal power state of the disk. If such information is not present, then it could drop a hint to a power manager regarding what the ideal power state should have been for the device. After receiving several such *ghost hints*, the power manager can alter the power state of the device [2].

5.3.2 DVFS Under Application Control

So far, we have seen DVFS schemes being implemented either by the operating system or by the hardware itself. In both cases, the decisions have to be taken not on the basis of future requirements of the application but on the basis of past observed workload history. However, since the power-performance requirements of different applications are distinct, power management policies that are tailor made for the applications could result in improved power efficiency with minimum effect on performance [28]. A few example applications having varying nature of operation and the associated unique power management strategies are discussed below.

5.3.2.1 MPEG Video Decoder

MPEG video decoder is a soft real-time application – it needs to meet timeliness constrains, failing which, the quality of the user experience is degraded. Other applications such as DVD playback, audio players, music synthesizers, and video capture belong to the same class of soft real-time applications. These applications could be abstracted as a sequence of tasks such that each task completes within a given time. Applications in this class could use the following DVFS policy.

Consider a task among the sequence of tasks needed to be executed by the application. Let the task completion deadline be *d* starting from time *t*. If *c* is the CPU time needed to complete the task when the CPU is operated at maximum frequency and *e* is the CPU time allotted to this task before the deadline, the processor speed is calculated as shown below.

1. If $t + c > d$, the task is bound to miss the deadline even when operated at maximum frequency. Hence, we choose to operate at maximum CPU frequency.
2. If $e < c$, the CPU demand exceeds its availability and the task is bound to miss the deadline in this case also. Hence, it is best to run the processor at maximum frequency.
3. If $t + c < d$ and $e > c$, the task can be slowed down so that it completes just at the deadline. The frequency f at which the CPU is to be operated is calculated as

$$f = \frac{c}{\text{min(e,d-t)}} \times f\text{max} \tag{5.2}$$

In order to compute f, the application needs to know the CPU availability *e* and an estimate of the processor demand c.

Estimation of CPU availability: An interface could be defined between the application and the OS such that the application receives the start and end times of, say, the previous *k* instances when this application was scheduled on the CPU. The average of times allotted in these previous instances can be used as an estimate for the availability in the next instance of the same application scheduled on the CPU.

Estimation of CPU demand: This could be obtained by characterizing the workload of a task in the application with respect to parameters that are expected to vary from task to task. For example, in the case of MPEG decoder, the decode time of a frame is found to be varying as a function of frame size and type of frame (frames are of three types I,B and P). Hence, a predictor could be built that computes the workload of the frame as a function of size and type of the frame. The predictor stores the observed decode times of previous *N* frames at full CPU frequency, to refit the prediction function to the parameters – size and type. Since each frame is annotated with a header that contains the information of frame size and type of frame, prior to decoding a frame we can extract this information and obtain the workload estimate from the prediction function.

5.3.2.2 Word Processor

This is an example of an interactive real time application. Several other applications such as games and web browsers, fall in this category of applications. These applications also consist of tasks that are to be finished within a deadline, but the tasks are initiated by an event of user interaction with the application. Hence, the workload of the applications is to be characterized as a function of these events. Since the number of these events types is generally very large, a reasonable workload characterization is not feasible for such applications. Instead, an approach of gradually increasing the CPU frequency to satisfy the CPU demand can be employed for power management of these applications. The duration available for the task is divided into a number of sub-intervals. Processing is started at minimum CPU frequency and every time a sub-interval is crossed before task completion, the CPU frequency is scaled to next available frequency level.

5.3.2.3 Batch Compilation

Compilation using *make* is a batch application, where throughput is more important than the time taken for completion of individual tasks – in this case, a task being compilation of a program. Since it is difficult to estimate the compilation time of each program, the best strategy in this case would be to allow the end user to specify the required speed settings. For example, the user can specify the priority of the batch application to be low, and hence cause it to run in the background.

Thus we see that, different DVFS policies suit different applications, and the application programmer can contribute significantly to efficient power management depending on the power-performance characteristics of the application. As mentioned earlier, an enhanced interface is necessary through which the application can collect resource utilization statistics from the OS. Secondly, the scheduler should be modified such that per-process CPU power settings are maintained and conveyed to the underlying hardware whenever the program is scheduled for execution. Finally, the OS needs to have the ability to map the application-specific power setting to the appropriate CPU frequency supported by the hardware.

5.3.3 Output Quality Trade-offs

Often, applications have multiple choices of solutions at their disposal for a certain processing task. Different algorithms with different computational complexities could be employed for the same processing task, with different associated quality of results. Such choices could be judiciously exercised by an application when it is made aware of the status of resources in the run-time environment. For example, an MPEG encoder under power constraints could sacrifice compression efficiency by skipping some steps in encoding process. Of course, the trade-offs involved here – less energy to encode vs. more energy due to possibly larger I/O – should be properly studied before making the decision. Similarly, a video player with access to multiple versions of videos with different image sizes, could select smaller images when under energy constraints [44].

Many applications in the signal processing and graphics domain are characterized by a graceful degradation feature with respect to the bit-width of data types used for computation. Such flexibilities can be exploited by applications to continue operation with reduced quality of output when under power and energy constraints. For example, when battery life-related constraints do not permit full-fledged processing with double precision arithmetic, an application could continue to operate by converting data to single precision and operating upon it, or by shifting to fixed point arithmetic.

5.4 Summary

Once power saving mechanisms have been incorporated into the underlying hardware, appropriate hooks need to be provided so that the software executing on the system can fully exploit them. In this chapter we covered the software components that can benefit from power awareness: the operating system, the compiler, and application software.

When the system under consideration is extended to include multiple tasks and multiple components such as the CPU, memory, I/O devices, and other resources, it is clear that the operating system emerges as an attractive entity in which to perform power management actions, since it has a good overall view of the resource usage by the different system tasks. We outlined several power management techniques including the important concept of intra-task and inter-task dynamic voltage and frequency scaling for real-time and non-real time systems.

The compiler interface is directly affected the first that is affected by the new hardware feature, since the compiler generates the code to execute on the hardware. Since a compiler has a deeper view of the program that is to ultimately execute on a processor, it can take power management decisions that may be difficult to handle at run time. We discussed different power optimization mechanisms involving the compiler: loop transformations, instruction encoding and scheduling, compilation for dual instruction architectures, instruction set extension, compiler directed power gating, and finally, disk optimizations. Finally, the application program can be made aware of the different hooks and knobs provided by the run-time environment to enable close monitoring of the state of system resources, as well as passing on crucial hints to the operating system about the state of the application.

References

1. Advanced RISC Machines Ltd: ARM7TDMI (Rev 4) Technical Reference Manual
2. Anand, M., Nightingale, E.B., Flinn, J.: Ghosts in the machine: interfaces for better power management. In: MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services, pp. 23–35 (2004). DOI http://doi.acm.org/10.1145/990064.990070
3. ARC Cores: ARCtangent-A5 Microprocessor Technical Manual
4. Azevedo, A., Issenin, I., Cornea, R., Gupta, R., Dutt, N., Veidenbaum, A., Nicolau, A.: Profile-based dynamic voltage scheduling using program checkpoints. In: DATE '02: Proceedings of the conference on Design, automation and test in Europe, p. 168. IEEE Computer Society, Washington, DC, USA (2002)

5. Biswas, P., Banerjee, S., Dutt, N., Pozzi, L., Ienne, P.: ISEGEN: Generation of high-quality instruction set extensions by iterative improvement. In: DATE '05: Proceedings of the conference on Design, Automation and Test in Europe, pp. 1246–1251. IEEE Computer Society, Washington, DC, USA (2005). DOI http://dx.doi.org/10.1109/DATE.2005.191
6. Bona, A., Sami, M., Sciuto, D., Zaccaria, V., Silvano, C., Zafalon, R.: Energy estimation and optimization of embedded vliw processors based on instruction clustering. In: DAC '02: Proceedings of the 39th conference on Design automation, pp. 886–891. New Orleans, Louisiana, USA (2002)
7. Burd, T.D., Brodersen, R.W.: Design issues for dynamic voltage scaling. In: ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design, pp. 9–14. ACM, New York, NY, USA (2000). DOI http://doi.acm.org/10.1145/344166.344181
8. Butts, J.A., Sohi, G.S.: A static power model for architects. In: Micro33, pp. 191–201 (2000). URL citeseer.ist.psu.edu/butts00static.html
9. Choi, K., Soma, R., Pedram, M.: Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times. IEEE Transactions on CAD **24**(1), 18–28 (2005)
10. Clark, L.T., Hoffman, E.J., Biyani, M., Liao, Y., Strazdus, S., Morrow, M., Velarde, K.E., Yarch, M.A.: An embedded 32-b microprocessor core for low-power and high-performance applications. IEEE Journal of Solid State Circuits **36**(11), 1599–1608 (2001)
11. Clark, N., Zhong, H., Mahlke, S.: Processor acceleration through automated instruction set customization. In: In MICRO, pp. 129–140 (2003)
12. Deeney, J.: Reducing power in high-performance microprocessors. In: International Symposium on Microelectronics (2002)
13. Dehnert, J.C., Grant, B.K., Banning, J.P., Johnson, R., Kistler, T., Klaiber, A., Mattson, J.: The transmeta code morphing^{TM}software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In: Proceedings of the international symposium on Code generation and optimization, pp. 15–24 (2003)
14. Govil, K., Chan, E., Wasserman, H.: Comparing algorithm for dynamic speed-setting of a low-power cpu. In: MOBICOM, pp. 13–25 (1995)
15. Gurumurthi, S., Sivasubramaniam, A., Kandemir, M.T., Franke, H.: Drpm: Dynamic speed control for power mangagement in server class disks. In: 30th International Symposium on Computer Architecture, pp. 169–179 (2003)
16. Hewlett-Packard, Intel, Microsoft, Phoenix Technologies Ltd., and Toshiba: Advanced Configuration and Power Interface Specification (2009)
17. Intel Corporation, http://www.intel.com/design/iio/manuals/273411.htm: Intel 80200 Processor based on Intel XScale Microarchitecture
18. Intel Corporation, http://www.intel.com/design/intelxscale/273473.htm: Intel XScale(R) Core: Developer's Manual
19. Ishihara, T., Yasuura, H.: Voltage scheduling problem for dynamically variable voltage processors. In: Proceedings of the 1998 International Symposium on Low Power Electronics and Design, 1998, Monterey, California, USA, August 10-12, 1998, pp. 197–202 (1998)
20. Jejurikar, R., Pereira, C., Gupta, R.K.: Leakage aware dynamic voltage scaling for real-time embedded systems. In: Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004, pp. 275–280 (2004)
21. Jiang, H., Marek-Sadowska, M., Nassif, S.R.: Benefits and costs of power-gating technique. In: ICCD '05: Proceedings of the 2005 International Conference on Computer Design. IEEE Computer Society, Washington, DC, USA (2005)
22. Kandemir, M., Son, S.W., Chen, G.: An evaluation of code and data optimizations in the context of disk power reduction. In: ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design, pp. 209–214. San Diego, CA, USA (2005)
23. Kandemir, M., Vijaykrishnan, N., Irwin, M.J., Ye, W.: Influence of compiler optimizations on system power. In: Proceedings of the 37th Design Automation Conference, pp. 304–307. Los Angeles, USA (2000)

24. Kim, C.H., Roy, K., Hsu, S., Krishnamurthy, R., Borkar, S.: A Process Variation Compensating Technique with an On-Die Leakage Current Sensor for nanometer Scale Dynamic Circuits. IEEE Transactions on VLSI **14**(6), 646–649 (2006)
25. Kim, T.: Application-driven low-power techniques using dynamic voltage scaling. In: 12th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2006), 16-18 August 2006, Sydney, Australia, pp. 199–206 (2006)
26. Lee, C., Lee, J.K., Hwang, T., Tsai, S.C.: Compiler optimization on vliw instruction scheduling for low power. ACM Trans. Des. Autom. Electron. Syst. **8**(2), 252–268 (2003)
27. Liao, W., He, L., Lepak, K.: Ptscalar version 1.0 (2004). URL http://eda.ee.ucla.edu/PTscalar/
28. Liu, X., Shenoy, P., Corner, M.D.: Chameleon: Application-level power management. IEEE Transactions on Mobile Computing **7**(8), 995–1010 (2008). DOI http://dx.doi.org/10.1109/TMC.2007.70767
29. LSI LOGIC: TinyRISC LR4102 Microprocessor Technical Manual
30. Mahesri, A., Vardhan, V.: Power consumption breakdown on a modern laptop. In: Power-Aware Computer Systems, pp. 165–180 (2004)
31. Mehta, H., Owens, R.M., Irwin, M.J., Chen, R., Ghosh, D.: Techniques for low energy software. In: ISLPED '97: Proceedings of the 1997 international symposium on Low power electronics and design, pp. 72–75. Monterey, USA (1997)
32. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufman, San Francisco, CA (1997)
33. Petrov, P., Orailoglu, A.: Compiler-based register name adjustment for low-power embedded processors. In: ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design, p. 523 (2003)
34. Powell, M., Yang, S.H., Falsafi, B., Roy, K., Vijaykumar, T.N.: Gated-vdd: a circuit technique to reduce leakage in deep-submicron cache memories. In: ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design, pp. 90–95 (2000)
35. Sery, G., Borkar, S., De, V.: Life is cmos: why chase the life after? In: DAC '02: Proceedings of the 39th annual Design Automation Conference, pp. 78–83. ACM, New York, NY, USA (2002). DOI http://doi.acm.org/10.1145/513918.513941
36. Shao, Z., Xiao, B., Xue, C., Zhuge, Q., Sha, E.H.M.: Loop scheduling with timing and switching-activity minimization for vliw dsp. ACM Trans. Des. Autom. Electron. Syst. **11**(1), 165–185 (2006)
37. Shin, D., Kim, J., Lee, S.: Intra-task voltage scheduling for low-energy, hard real-time applications. IEEE Design & Test of Computers **18**(2), 20–30 (2001)
38. ST Microelectronics: ST100 Technical Manual
39. Su, C.L., Despain, A.M.: Cache design trade-offs for power and performance optimization: a case study. In: ISLPED '95: Proceedings of the 1995 international symposium on Low power design, pp. 63–68. ACM Press, New York, NY, USA (1995)
40. Sun, F., Ravi, S., Raghunathan, A., Jha, N.K.: Synthesis of custom processors based on extensible platforms. In: ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design, pp. 641–648. ACM, New York, NY, USA (2002). DOI http://doi.acm.org/10.1145/774572.774667
41. Tomiyama, H., Ishihara, T., Inoue, A., Yasuura, H.: Instruction scheduling for power reduction in processor-based system design. In: DATE '98: Proceedings of the conference on Design, automation and test in Europe, pp. 855–860. Le Palais des Congr&#233;s de Paris, France (1998)
42. Tschanz, J.W., Narendra, S.G., Ye, Y., Bloechel, B.A., Borkar, S., De, V.: Dynamic sleep transistor and body bias for active leakage power control of microprocessors. IEEE Journal of Solid State Circuits **38** (2003)
43. Valluri, M., John, L., Hanson, H.: Exploiting compiler-generated schedules for energy savings in high-performance processors. In: ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design, pp. 414–419. ACM Press, New York, NY, USA (2003)
44. Venkatachalam, V., Franz, M.: Power reduction techniques for microprocessor systems. ACM Computing Surveys **37**(3), 195–237 (2005)

45. Weiser, M., Welch, B.B., Demers, A.J., Shenker, S.: Scheduling for reduced cpu energy. In: OSDI, pp. 13–23 (1994)
46. Yao, F.F., Demers, A.J., Shenker, S.: A scheduling model for reduced cpu energy. In: FOCS, pp. 374–382 (1995)
47. Yun, H.S., Kim, J.: Power-aware modulo scheduling for high-performance vliw processors. In: ISLPED '01: Proceedings of the 2001 international symposium on Low power electronics and design, pp. 40–45. Huntington Beach, USA (2001)

Chapter 6
Power Issues in Servers and Data Centers

Power optimization as a research topic was first studied in the context of portable and handheld systems where saving battery life was of prime importance. However, since that time, the need for saving power has become significantly more pervasive all over the computing machinery, from portable devices to high-end servers and data centers. The need for power optimization in larger scale computing environments such as servers and data centers arise from the increasing maintenance costs (including the electricity charges) due to the power demands of a very large number of computers. This introduces new contradicting requirements in the server design space, which, in the past, were designed to a different set of specifications. Performance was the primary design metric, with execution time and throughput being the main considerations. In addition, reliability and fault-tolerance related concerns were also significant, leading to designs with redundancy that offered high availability. While these concerns continue to be important, the emergence of power efficiency has led to interesting innovations in the way such systems are conceived, architected, and programmed.

The previous two decades have led to the emergence of data centers as integral components of the world's computing infrastructure. These facilities, consisting of thousands of powerful computers, are the hosting sites of a variety of computing services involving data storage, search, and processing. The concentration of a large number of powerful computers in a small area in such a facility is understood to lower the overall cost of providing the services, due to economies of scale in the management leading to operational efficiencies. At the same time, this consolidation also leads to large amounts of power being consumed by the data centers, clearly paving the way for innovation in energy efficiency in the domains of relatively powerful server computers, clusters of servers, and entire data centers [1].

6.1 Power Efficiency Challenges

In what way are power and energy efficiency concerns at macro-level entities such as servers and data centers different from those at the chip-level? At the level of servers, the building blocks that play a role in determining and influencing energy

P.R. Panda et al., *Power-efficient System Design*, DOI 10.1007/978-1-4419-6388-8_6,

dissipation include not only devices that are already well researched for low-power implementation, such as processors and memories, but also other components such as fans, power supplies, network and I/O controllers, and disks. The operating system controlling the hardware devices plays a key role in the budgeting and management of power for the server as a whole. Due to the disparate nature of the building blocks which range from electronic to mechanical components, the energy optimization problem for the server is an interesting new challenge. Chip level power optimizations still need to be anticipated and the overarching scenarios such as power-performance trade-offs are similar, but the power management mechanisms cannot be trivially re-targeted. The time periods over which events occur are no longer in the nanoseconds but at least three orders of magnitude larger. The power models needed for optimization at the server level are similarly macroscopic in nature.

Clusters of servers and data centers represent the next higher level in the architectural hierarchy and energy management in this context involves co-ordinating the activity of a possibly large number of individual servers. Again, a sufficiently high level view of the constituent servers needs to be taken in order to do an intelligent power management within practically available periods of time, and subtle trade-offs exist between how much decision making should be done at a centralized level and how much autonomy to provide, for example, to individual CPUs for determining their own decisions dynamically.

Given a power budget, a power management strategy would involve the apportioning of the budget among the different components. However, several factors discussed below lead to an inefficient allocation.

6.1.1 Nameplate Power Overestimates Actual Power

The term *Nameplate Power* is used to indicate the power rating for a server system published by the manufacturer. The power drawn by the system is guaranteed to not exceed this number. While such a label could be useful, it is often computed by merely adding the maximum power dissipation advertised for the individual components of the system! This is the equivalent of estimating maximum power of a circuit by adding the maximum power drawn by each component separately, which of course is a gross overestimate because the different components are extremely unlikely to draw peak power simultaneously.

In a comparison of nameplate power against actual power drawn, a server with the component power ratings given in Table 6.1 was used [5]. The server consists of a motherboard with 2 CPUs, 4 memory modules, 2 PCI slots, and an IDE disk drive. The total power drawn by the system components is 213 W. With a power supply efficiency of 85%, the nameplate AC power drawn by the system is 251 W. When such a system was subjected to rigorous power measurement by power intensive benchmark test suites, the actual peak power observed was only 145 W, which is only 58% of the nameplate power. This is one illustrative example with relatively low-power CPUs. However, the observation is routine, with the actual percentage

Table 6.1 Rated power dissipation of server components [5] for relatively low power CPUs. AC power is higher to account for power supply inefficiency

Component	Power
2 CPUs	80 W
4 Memory modules	36 W
Disk	12 W
2 PCI slots	50 W
Motherboard	25 W
Fan	10 W
Total	213 W
AC power	251 W

varying across systems. This calls into question the utility of nameplate power as a serious metric for use in power management strategies. A separate calibration is typically necessary to measure peak power in more carefully controlled environments, executing programs spanning the range of applications expected to run on the server.

6.1.2 Installed vs. Utilized Capacity

Estimates for data center power are often based on the maximum possible occupancy, which is seldom true in practice. In reality, a data center's computing equipment grows over a period of time, leading to the actual power dissipation being much less than the budgeted power.

Redundancies built into the power supply design also leads us in the same direction. Dual power supplies are commonly used in computer equipment so that one can serve as a back up if the other fails. Since each power supply is connected to its own circuit, the provisioned power is clearly double that of the maximum dissipated power [11]. Such overestimation ultimately leads to excess cooling capacity, leading to higher cost of operation of the data center. Further, racks in data centers are also frequently not filled up completely because of basic capacity mismatches. For example, if we have 300W servers and 1KW racks, then one rack will house only 3 servers, making the total peak power dissipation 900W, leaving 10% power unutilized in the rack.

6.1.3 Load Variation

Power drawn by server depends on the work being currently performed, and hence is inherently a dynamic quantity. When the server is idle, its dissipated power is much less than its peak power. Thus, it is clear that the dynamic behavior of the servers in the data center has to be taken into account in any intelligent power management strategy spanning clusters and data centers. Similarly, at a higher level of the hierarchy, work done in different servers may not be correlated and all may not draw their respective peak power simultaneously. Thus, it would be inaccurate to estimate the data center power as the sum of the rated power for the constituent servers, analogous to the issue with nameplate power.

Factors such as the ones mentioned above lead to overstating of power requirements for data centers, which has several undesirable consequences including higher infrastructure costs incurred by data center owners in terms of provision for cooling and excess generation and distribution for utility planners. These factors also point to possible innovations in power management policies for servers and data centers.

6.2 Where does the Power go?

Where does the power go in servers and data centers? It is worth examining the data for some example machines to understand the possible areas to be targeted by power management schemes. Table 6.2 shows a power dissipation comparison of two configurations of an IBM p670 server system. Column 2 shows the power dissipation in a "small configuration" version with a 4-way server consisting of 4 single-core processors with a 128 MB L3 cache and a 16 GB memory. Column 3 shows the numbers for a "large configuration" 16-way version consisting of a dual multi-chip module with 4 dual-core processors, 256 MB L3 cache, and 128 GB memory.

The components of the server power are as follows:

- Processors, caches, and cache controllers
- Off-chip memory, including L3 cache, DRAM, memory controllers
- I/O and other equipment
- Fans dedicated for cooling processors and memory
- Fans dedicated for cooling I/O components

Some of the data are approximate (large configuration data estimated by scaling up small configuration data, fan data read from data sheets, etc.), but nevertheless, useful in setting the context for where power management policies should concentrate. Processors and memory account for the bulk of the power dissipation, but there is a major difference in the relative power of the processor and memory among the two configurations. The trend of high-end servers being packed with increasing

Table 6.2 Power dissipation in components of small and large configuration servers [9]

Component	Small Configuration	Large Configuration
Processors and Caches	384	840
Memory	318	1223
I/O and others	90	90
CPU/Memory Fans	676	676
I/O Fans	144	144
Total	1614	2972

amounts of memory leads to memory power becoming the dominant component, pointing to the increasing importance of memory in server power optimizations.

The fans in the server are responsible for cooling the processor, memory, and other devices. The relatively high percentage of power attributed to the fans might seem surprising. The fans in Table 6.1 had a relatively lower wattage. However, servers dissipating higher power need more powerful fans that themselves require higher power.

Disk power is not directly accounted for in the above data, primarily because modern server systems have the disks decoupled from the compute unit and bundled into a separate networked storage. The storage system now becomes the target of a possibly independent power management strategy.

As expected from the above discussion, most power management strategies directly involve manipulation of the processor and memory activity. The move towards a large number of processor cores lends itself to power management control at the individual core level, where the activity state of the cores can be manipulated to range from fully active to shutdown, with other intermediate states also being available when necessary.

At the data center level, it is worth looking at the power losses along the delivery chain to the computing units to see where energy is dissipated even before reaching the computers. Figure 6.1 illustrates the inefficiencies of the power delivery chain

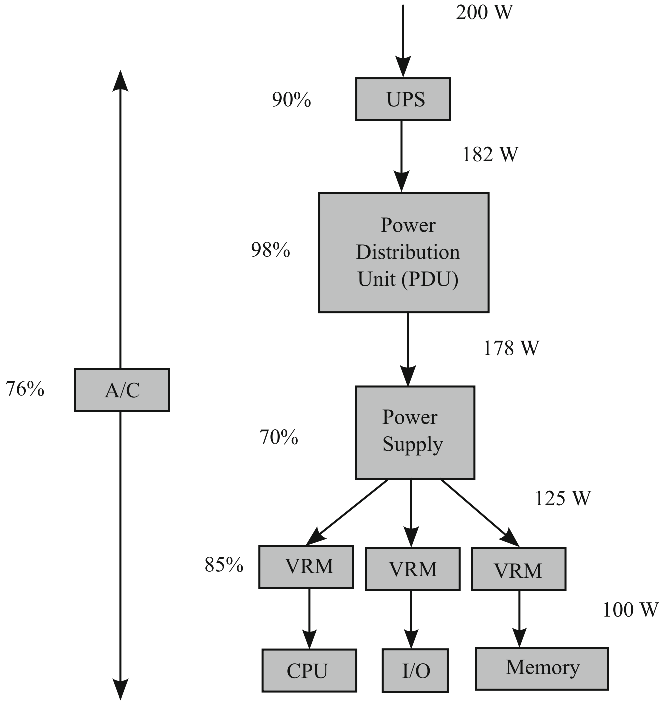


Fig. 6.1 Power losses along the delivery chain. The percentages indicate the efficiencies of conversion and distribution. Only 40% of the drawn power actually reaches the computation units

arising out of energy losses at various stages [10]. For a hypothetical loading of the server, the power supply unit converting AC to DC power within the server may operate at 70% efficiency (i.e., 30% of the power dissipated by the server is lost at the power supply unit itself). Voltage Regulator Modules (VRM) then convert the output DC into specific voltages needed for different chips. The processor, memory, peripherals and disk may need different voltages such as 12V, 3.3V, 5V, etc. The voltage conversion causes energy losses at the VRMs; only 85% of the power reaching the VRMs is forwarded to the chips. Beyond the rack of servers, the next hop is the Power Distribution Unit (PDU), which is generally considered very efficient at 98%. Powering the entire data centre is the Uninterruptible Power Supply, with a 90% efficiency. The heat generated by all the data center equipment has to be cooled by an air-conditioning unit whose energy requirements are proportional to the power dissipated inside the equipment, with a 76% efficiency.

The cumulative result of energy losses along the distribution chain is that only 40% (product of the efficiencies) of the energy drawn by the data center actually reaches the electronic hardware. The scale of power dissipated at the various distribution stages could pave the way for future power management strategies taking a more holistic view of the optimization problem, enlarging the emphasis to include mechanical and other components in the data center in addition to the electronics.

6.3 Server Power Modeling and Measurement

In order to perform power management functions at the server level, a sufficiently accurate power model is necessary. Such a model helps in the identification of optimization opportunities and in the prediction of the consequences of the decisions taken. An important criterion of a power model that could be useful at the server level is that the model should be characterized by easily accessible parameters. For example, an instruction set level power model for the processor is not suitable for use here because getting a trace of all instructions executed on the processor is expensive and would also interfere with the working of the processor.

The processor's power dissipation consists of a static and dynamic component, with the static component being approximately constant and the dynamic component varying with the activity of the processor. The static component is not strictly constant because the processor's internal power management policy may induce variations in the static power, but could still be treated as such at the high level. The extent of activity in the processor is a very complex function of many different simultaneous states and transitions. However, one gross metric that has been identified as a reasonable representation of dynamic activity is *CPU utilization*. The utilization metric could be interpreted in different ways, but a simple interpretation is *the fraction of clock cycles for which the CPU is NOT halted*. Such a metric could be used for the entire processor, or derived individually for the constituent cores in a multicore processor. CPU utilization is a very attractive metric from the system management point of view, because this parameter can be easily obtained through the operating system.

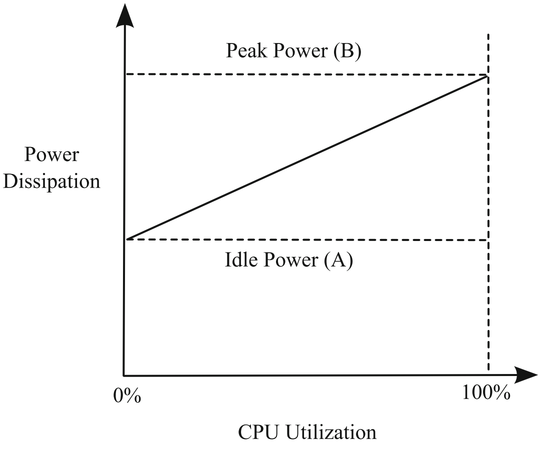


Fig. 6.2 Server power model based on CPU utilization. A linear model serves as a good approximation

A simple idealized power model for a processor is shown in Fig. 6.2. Point A represents the *idle power* at 0% utilization, when the processor is doing nothing. Point B represents the *peak power* drawn at 100% utilization. The power varies linearly with the utilization according to the equation:

$$\text{Power} = A + (B - A) \times \text{CPU\_utilization} \tag{6.1}$$

where *CPU_utilization* is a fraction between 0 and 1. The power numbers A and B are usually obtained by an initial calibration. Although this model is very simple and approximate, studies show that it actually tracks the CPU power well enough to be practically useful, in spite of ignoring the details of the processor's internal activity [5]. For a data center, the model above generalizes to A representing the idle power of all servers combined and B representing the total power when all servers are 100% busy. CPU utilization would now represent the average utilization across all processors.

More complex server power models could be built by accessing various performance counters that are built into modern processors. Going down one level in detail, we can keep track of the relative activity rates for the CPU, memory, network, and hard disk. A high-level power model could be now based on the following data:

- CPU utilization
- Memory access count
- Hard disk access rate
- Network access rate

The power equation is a linear combination of the above factors:

$$\begin{aligned}\text{Power} = {} & A \times \text{CPU\_utilization} + B \times \text{memory\_access\_count} + \\ & C \times \text{disk\_access\_rate} + D \times \text{network\_access\_rate} + E\end{aligned}$$

For different systems, an initial calibration is necessary to obtain the constants A, B, C, D, and E [3]. The power estimate obtained from the above model also closely tracks the measured power. The reason CPU utilization performs well is that the other server components can also be thought of as being dependent on CPU activity. For example, if more instructions are executed from the CPU, then more of them are fetched from the memory, which also increases the memory access rate, leading to a good correlation between the CPU utilization and the other activity metrics.

6.4 Server Power Management

In this section we will review some power management techniques that are applicable to single server computers, possibly employing a multiprocessor architecture.

6.4.1 Frequency Scaling

System level power management is achieved in several server systems through controlling the voltage and/or frequency settings. In several systems, the two parameters are not independently adjustable, but we have the choice of different *power states*, with the low power states corresponding to lower voltage and frequency, and the high performance states corresponding to higher voltage and frequency. Sometimes, the control available is only in terms the system clock frequency, with the voltage being fixed. *Clock throttling* is a simpler mechanism to control the performance state without changing either voltage or frequency. The clock is merely gated for some period at regular intervals to reduce the rate of activity, thereby reducing the average dynamic power. In multicore systems, ideally we should have independent voltage and frequency control over each of the processor cores, but even a relatively modern multicore processor such as the POWER7 [15] does not provide this flexibility; only the frequency of the individual cores is independently controllable.

The choice of the performance and power states in a server should be made judiciously by considering the overall priorities.

Objective 1: deliver highest speed. If delivering the fastest execution and response times is the primary goal, then the highest performance state is chosen.

Objective 2: deliver highest speed under power constraint. If delivering the fastest response while meeting a power budget is the goal, then the highest performance state that does not exceed this budget is chosen. The system would first need to be characterized for power dissipation at the different performance states.

Objective 3: deliver a given performance at lowest energy. If the goal is to minimize the total energy dissipated while delivering a particular performance

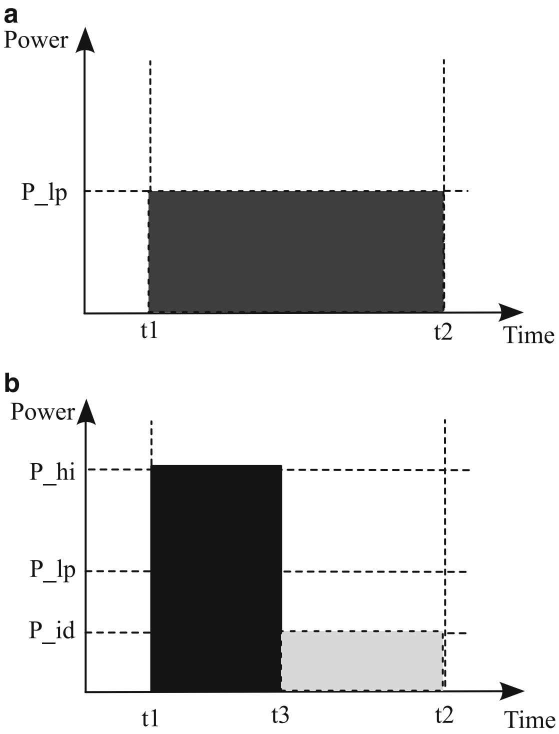


Fig. 6.3 Servers running at high performance and low power operating points. (**a**) Low power mode: the system is run at power P_lp, just meeting the deadline. (**b**) High performance mode: the system is run at high performance mode with power P_hi; it finishes faster, then idles at power P_id. Relative energy efficiency is determined by the relative power values

(measured in, say, throughput of tasks per unit time), then the choice is not obvious, and depends on the relative performance and power numbers of the different states [12].

Consider a server system that is required to execute a given load within a given time period. The start time is $t1$ and the deadline is $t2$ (Fig. 6.3). We have the choice to run the server in two different modes: low power mode dissipating P_lp power, and high performance mode dissipating P_hi power. Assume that the system is able to meet the $t2$ deadline when run in the low power mode, as shown in Fig. 6.3(a). The energy consumed in this case is given by:

$$E\_lp = P\_lp \times (t2 - t1) \tag{6.2}$$

Alternatively, let us run the system in a high performance mode, leading to a faster execution. The execution completes at time $t3$ ($< t2$). For the remaining duration until $t2$, the system is idle, dissipating power P_id. The energy dissipated is given by:

$$E\_hi = P\_hi \times (t3 - t1) + P\_id \times (t2 - t3) \tag{6.3}$$

The deadline is met in both cases. Which alternative dissipates lower total energy? The relative values of E_lp and E_hi depend on the actual power numbers (P_hi, P_lp, and P_id), and execution times ($t2 - t1$ and $t3 - t1$). This problem generalizes to the following problem:

Problem: *Given a workload and a performance constraint, select the appropriate performance state/frequency at which the server system should be executed, so that the total energy dissipated is minimized.*

In general, we have to choose one among several different performance states or frequencies. Let us make the following definitions:

f_{min} The minimum frequency at which the system can be executed.
f_{max} The maximum frequency at which the system can be executed.
P_f The power dissipated when we execute the system at frequency f.
P_{idle} The power dissipated when the system is idle.

Let us assume that the system is able to just meet the deadline at the minimum frequency f_{min}, and the execution time is T_{min}. The energy corresponding to the execution of the system at this frequency is:

$$E_{min} = P_{min} \times T_{min} \tag{6.4}$$

We compare the above energy number with that of the total energy dissipated when the system is run at higher frequencies, with the same deadline T_{min}. We use a simple linear model of performance and power scaling – the power is a linear function of frequency (voltage and switching activity remaining constant), and execution time is inversely proportional to frequency. At frequency f, the execution time is:

$$T_f = T_{f_{min}} \times \frac{f_{min}}{f} \tag{6.5}$$

The system is idle for the remaining time = $T_{min} - T_f$. Thus, the total energy dissipation at frequency f is:

$$\begin{aligned} E_f &= P_f \times T_f + P_{idle} \times \left(T_{min} - T_f\right) \\ &= P_f \times \left(T_{f_{min}} \times \frac{f_{min}}{f}\right) + P_{idle} \times T_{f_{min}} \left(1 - \frac{f_{min}}{f}\right) \end{aligned}$$

Using the linear power model, we have the power at frequency f given by:

$$P_f = P_{f_{min}} + m \times (f - f_{min}) \tag{6.6}$$

where the slope m is determined by the characteristics of the processor. Substituting into the expression for E_f, we obtain:

$$E_f = \left(P_{f_{min}} + m \times (f - f_{min})\right) \times \left(T_{f_{min}} \times \frac{f_{min}}{f}\right) + P_{idle} \times T_{f_{min}} \left(1 - \frac{f_{min}}{f}\right) \tag{6.7}$$

Thus, the choice of frequency f for energy minimization can be based on the computed value E_f at that frequency. This computation is determined by:

1. nature of the application.
2. characteristics of the processor.

The nature of the application exhibits itself in the execution time estimate used in the above computation. The T_f value computed above is valid for CPU-intensive workloads, but not in situations where other components such as memory or disk are the bottleneck. For example, if memory is the bottleneck, when we increase the frequency, then memory requests will require a *higher* number of cycles than at a lower frequency. This increases T_f beyond $T_{f_{min}} \times \frac{f_{min}}{f}$, which had assumed that the number of clock cycles remains the same.

Similarly, the internal design of the processor, which determines the value of m above, also influences the decision of whether it is energy-efficient to run the system at frequency f. Experiments on servers based on an 850 MHz Pentium and a PowerPC 405GP revealed that the Pentium gives the lowest energy at the highest performance state (frequency), while the PowerPC dissipates the lowest energy at the lowest performance state that meets the time constraint. Thus, the behavior of the two systems is opposite, and power management algorithms have to take this difference into account, in addition to the difference in the nature of the applications [12].

Where the option of voltage scaling is available, an analysis similar to the above is necessary. Since power varies as the square of the voltage, the scaled power does not vary linearly. However, note that only the CPU power may vary directly with frequency change. The voltages at the other components are likely to be the same. Hence, a separate characterization of both the server and the application is required to estimate the impact of voltage and frequency variations.

6.4.2 *Processor and Memory Packing*

It is instructive to look at the workload variation in some commercial server applications in order to understand the opportunities for optimization. Figure 6.4 gives one view of the typical load seen by a transaction processing server processing web service or data base requests in a commercial enterprise [9]. The period with relatively high load corresponds to the office hours in a business. The load is lower outside business hours, especially late night to early morning.

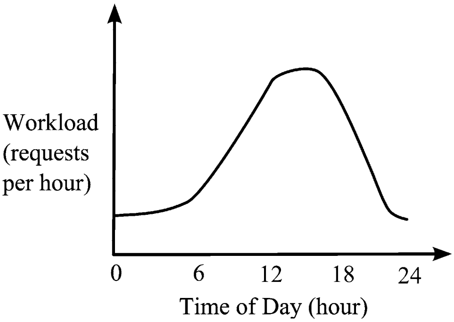


Fig. 6.4 Workload variation over a 1-day period in a commercial server. The significant variation creates power optimization opportunities

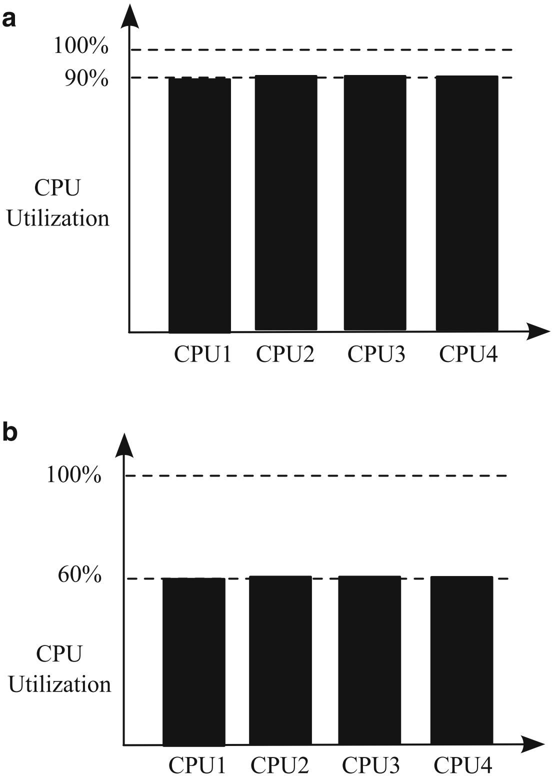


Fig. 6.5 Balancing the workload in multiprocessor (SMP) servers. Performance is maximized when the servers are equally loaded in both heavy and light load conditions. (**a**) Heavy Load. (**b**) Light Load: Equally Balanced

Servers with a symmetric multiprocessor (SMP) or multicore architecture can effect energy optimizations by exploiting the situation above where wide variations exist in the workloads submitted to the server. Figure 6.5 shows two situations with heavy and light load faced by an SMP server with 4 processors. In Fig. 6.5(a), the

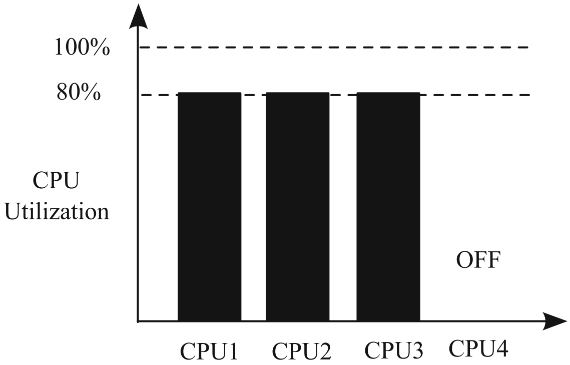


Fig. 6.6 Processor packing: asymmetric load distribution. Light load is distributed into a smaller number (3) of processors, switching one processor off

load is heavy and when distributed equally among the 4 processors, keeps all of them at 90% utilization. In Fig. 6.5(b), the load is lighter, and when distributed equally among the 4 servers, keeps all of them 60% utilized.

In contrast, Fig. 6.6 shows the same light load (of Fig. 6.5(b)) distributed over only 3 of the 4 processors. This causes the load on the 3 servers to increase to 80%, but gives us the option to turn the fourth server off. This is an example of an important class of optimizations called *Processor Packing* or Processor Consolidation [9]. An idle processor, as mentioned earlier, consumes a significant amount of power – in typical servers, it could dissipate 50-60% of the peak power consumed. Similarly, when processors are lightly loaded, their energy efficiency is low. Processor packing aims to produce an asymmetric load distribution, creating the potential for energy reduction by turning unused processors off. The power consumed by three CPUs operating at 80% utilization is expected to be smaller than that consumed by 4 CPUs operating at 60% utilization, though the exact power numbers depend on the actual processors used.

The concept of *memory packing* is analogous to the processor packing idea. DRAM memory modules with no active data can be dynamically set to low power mode so as to save power. When they are actually needed, the modules can be restored to active mode to enable their usage. Whether a memory module is active or idle depends on the data distribution. The data distribution can be influenced by the operating system's page allocation policies. A power aware page allocation policy would allocate new memory pages in memory modules that are already active, which reduces the number of active modules. Power can also be saved by periodic *page migration* by moving active data from one memory module to another so as to enable setting one of them to low power mode. The analysis is similar to the on-chip memory case discussed in Section 4.5, but the data objects are at a higher level of granularity. A similar view can also be taken during the memory address mapping phase. Multiple memory controllers may be present in the system, with a different set of banks assigned to each controller. If the allocated memory pages can be clustered into a small number of banks, one or more memory devices and controllers may be powered down.

Memory packing has associated with it a trade-off between energy efficiency and the delivered performance/bandwidth. Turning banks and controllers saves energy but leads to latency delays when requests are directed at banks that are turned off.

6.4.3 Power Shifting

Although the frequency scaling and clock throttling mechanisms help control the performance and power states of a server, a more fine-grain, independent control of the different server components could help exploit more energy efficiency opportunities. As seen in Section 6.3, the power model of a server could be improved by considering other parameters in addition to CPU utilization, such as memory access rate and disk access rate. We can attempt to control the processor and memory separately, since these two are the major power consuming components in any server.

The *Power Shifting* technique attempts to trade-off processor power for memory power in an attempt to deliver maximum performance under a given total power budget [6]. In an experiment conducted by running benchmark examples over a period of time, it was observed that the peak CPU power was 53W, and the peak memory power was 29W (Fig. 6.7). However, the peak total power was only 58W (as opposed to the sum of peak power for CPU and memory, which is 82W). This suggests that the CPU and memory are not drawing peak power at the same time, and that a dynamic apportioning of a total power budget between CPU and memory would achieve better performance than a static partition.

For controlling the power consumption of the CPU and memory components separately, throttling mechanisms are used. A characterization phase derives the linear

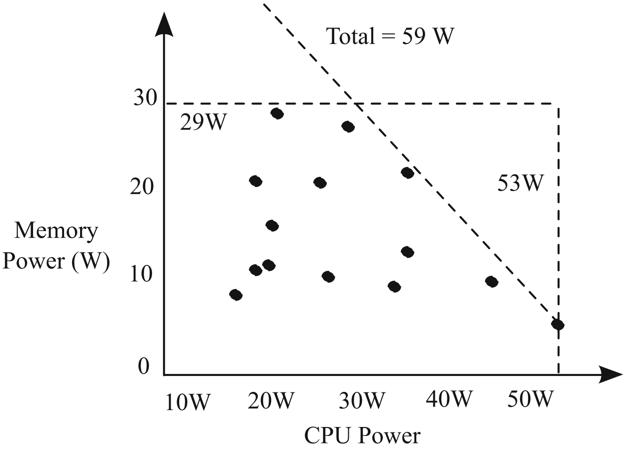


Fig. 6.7 Plot of CPU vs memory power at different times [6]. Sum of CPU power and memory power is always less than 58W (less than sum of peak CPU and memory power numbers)

relationship between power consumption and activity factors, measured as the number of operations in a time interval (instructions in the case of CPU and memory accesses in the case of memory). For throttling, hard limits are placed on the number of operations in every time interval. Once the limit is reached, the system is stalled until the start of the next interval. Larger or smaller values for the interval lead to coarse grain or fine grain control.

Different policies could be used to control the power allocation of the different components.

Static Allocation. In this case, we allocate the total power budget according to the ratio of the characterized peak power values for the CPU and memory. For example, if the peak CPU power is 100W and peak memory power is 50W, and a power budget of 90W is imposed, then we allocate power in a 2:1 ratio for CPU and memory, i.e., CPU gets 60W and memory gets 30W. Note that the example in Fig. 6.7 shows that this is inefficient, since the peaks for CPU and memory may not occur together, and hence, the total power in this approach may never come close to 90W. In the static allocation scheme, the budgets are fixed once and never modified during program execution.

Proportional by Last Interval (PLI). In this mechanism, it is predicted that the ratio of CPU and memory activity in the current interval will be the same as that in the next interval, and the power budgets are fixed accordingly. It is recognized that the power dissipation has an activity-dependent part and an activity-independent part. The activity-independent part is always budgeted for the components. The activity-dependent part is determined by the allocated activity in the next interval.

Sliding Window. This policy generalizes the prediction mechanism in PLI to include historical behavior from more than one previous interval. Here, the throttling decisions are taken every interval, but the monitoring period includes a window several consecutive intervals. The activity of the next interval is now predicted to be the average activity of the last several intervals. This leads to a smoothing out of transient departures from normal behavior.

Several other activity prediction mechanisms popularly used in system-level power optimizations could also be used for the power shifting process. Other variants include an *on-demand* strategy that does not interfere with the normal program working if the current system power is significantly below the budget, but switches to a PLI-like scheme when the power approaches the budget.

6.5 Cluster and Data Center Power Management

We will cover in this section the strategies applicable to server clusters and data centers, where the number of servers could range from a few computers (say within a single rack of servers) to hundreds of thousands of computers (in a large scale commercial data center). The basic power management mechanisms at the multiprocessor server level are also relevant at the server cluster and data center level, with the CPU core being substituted by an entire server. However, there are significant differences in the details of the parameters monitored, manipulation of server state, and the time intervals over which optimization is performed.

6.5.1 Power Capping/Thresholding

Since an idle server may dissipate 60% of the peak server power, it is expensive to maintain servers in idle state. This basic observation can be used to influence power management decisions at the cluster level. Similar to the processor packing/consolidation proposal discussed in Section 6.4.1, the workload can be distributed intelligently across the cluster of servers. During periods of high load, the load can be balanced equally among the servers. During idle periods, the load can be concentrated into a smaller number of servers, giving us the opportunity to switch the others off or set them to a low power state.

Figure 6.8(a) shows an idealized profile of the expected variation in power as the load increases in performance-optimized and energy-optimized server clusters using the informal strategy outlined above. In the performance-optimized version where all servers are always in active state, the power dissipation at low loads is relatively high because idle servers dissipate significant power. In contrast, in the energy-optimized version, the low-load power is much smaller due to aggressive power management leading to several servers being turned off. As the load increases, the total power increases linearly, with the performance-optimized and energy-optimized lines exhibiting different slopes. At the highest load, all servers operate in the highest performance states in both cases and the total power values converge.

Figure 6.8(b) shows the corresponding performance variation, characterized in terms of latency. At low loads the energy-optimized cluster exhibits higher latency because the smaller number of active servers are loaded to a higher extent in comparison to the performance-optimized cluster. As the load increases, the latency numbers become similar because the two configurations begin to look similar [2].

Consider a homogeneous server cluster consisting of identical servers, with a load balancing switch that can be enhanced to include power awareness (Fig. 6.9). The operating system on each server monitors the load and communicates the information to the central switch, which maintains a list of currently active servers and the extent to which each is loaded.

A performance-optimized cluster will attempt to distribute the load equally among all the servers, so a new request will be forwarded to the server with the lowest current load. As shown in Fig. 6.10(a), the capacity of the cluster in terms of servicing the requests remains constant irrespective of the actual load seen, leading to excess capacity at low loads.

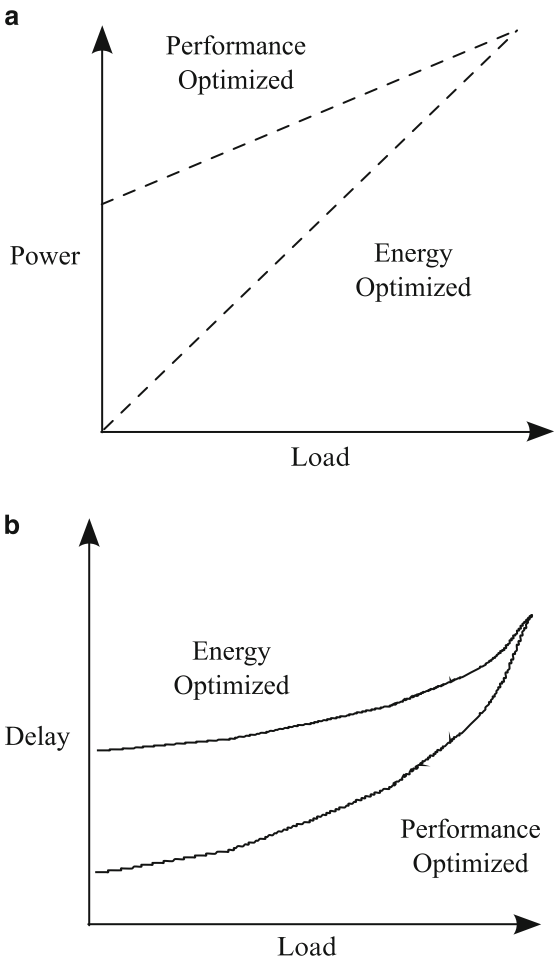


Fig. 6.8 Variation of power and delay in performance-optimized and energy-optimized server clusters. (**a**) At low load, the performance-optimized system consumes more power due to idle servers. At high load, all servers are heavily utilized, and power consumed is similar. (**b**) At low load, the performance-optimized system performs better because individual servers are lightly loaded; the energy-optimized system is worse because it uses fewer (heavily loaded) servers, so individual tasks have higher latency. At high load, server states are similar, so latencies are also similar

In an energy-optimized cluster, the switch will need the additional ability to add or remove nodes from the current active cluster. We can define a threshold T for the average utilization factor at which we wish to keep the set of active servers. During execution with k active servers, if the average utilization approaches T, then a new server is activated, making the total number of active servers $k + 1$. With k servers, the available capacity is kT, with capacity being measured in terms of CPU utilization multiples. When the load falls in the region of $(k - 1)T$, we

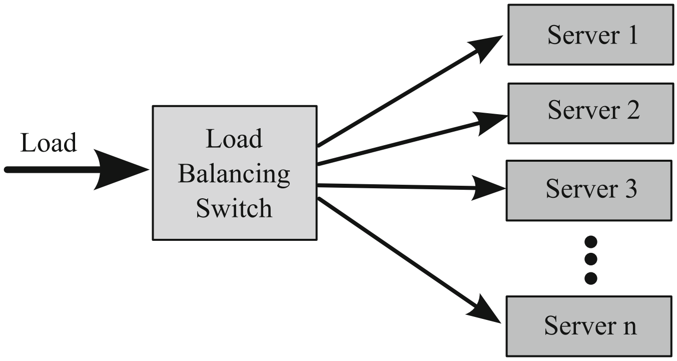


Fig. 6.9 Load balancing switch in server cluster. A central switch distributes tasks to one of *n* servers by keeping track of their individual states

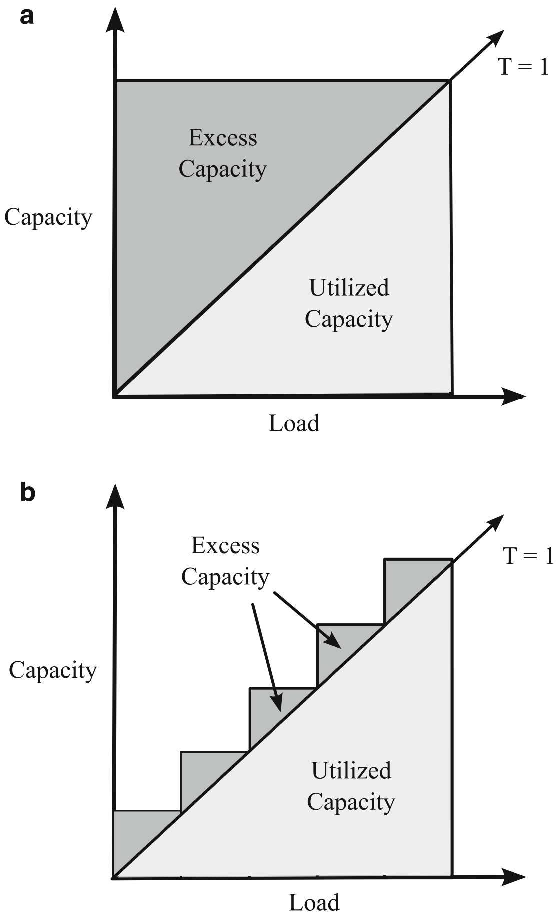
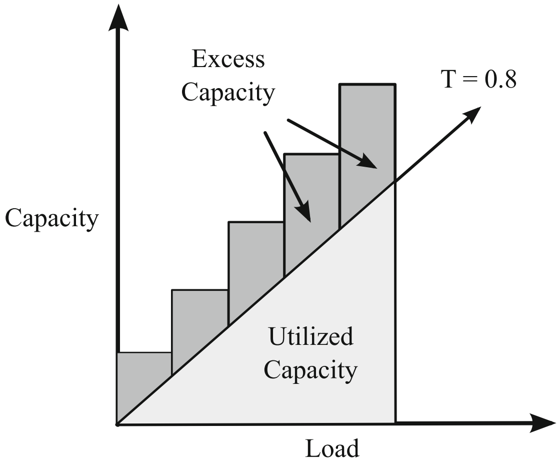


Fig. 6.10 Changing capacity with load in energy-optimized server cluster. (**a**) In performance-optimized system, all servers are always active, so the total capacity is constant. At low load, excess capacity is larger. (**b**) In energy-optimized system, new servers are turned on (i.e., extra capacity is added) as and when the load on the current set reaches 100% ($T = 1$), leading to a step-function like behavior in the capacity

Fig. 6.11 Capacity variation with a different threshold $T = 0.8$. New capacity is added when average load reaches 80% of current capacity



deactivate one server, decreasing the total number of active servers to $k - 1$. Thus, the available capacity changes as a step function, as shown in Fig. 6.10(b), where we use a threshold of $T = 1$. That is, a new server is commissioned when the existing capacity is fully utilized. Comparing with Fig. 6.10(a), this strategy leads to a significant reduction in excess capacity available, which is exploited to save power by turning the remaining servers off.

Figure 6.11 shows the scenario when we use a different threshold such as $T = 0.8$. Now, the load intervals are smaller – we switch server state at smaller intervals, and as shown by the shaded region, a greater excess capacity is maintained.

As seen above, the set of resources utilized by a cluster or data center can be dynamically varied depending on the current load. The task allocation decisions taken by the load distribution switch can be based either on actual measurement of instantaneous load, power, etc., or a performance/power model of the system in terms of high level parameters that can be tracked and manipulated by a load balancing unit. The decision to assign a task to a resource can be based on a more complex cost function that ultimately relates the cost of making the resource available for the task for that duration against the benefits obtained from allocating it. Thus, the provisioning of resources in a data center need not target the worst case utilization scenario, but can take into account a more nuanced analysis of the cost of not meeting the occasionally occurring peak request rate.

6.5.2 Voltage and Frequency Scaling

In addition to varying the set of currently managed active servers, the voltage scaling mechanism available on each individual server could also be used for cluster power management. When a specific voltage is selected for operation, the frequency is appropriately scaled so that the system still works reliably at the selected voltage. The general problem can be stated as follows.

Problem: *Given a workload, a performance constraint, and a set of n homogeneous servers that can be set to one of m different states, each setting corresponding to a voltage and frequency, select the appropriate setting V_i at which each individual server should be executed, so that the total energy dissipated is minimized.*

Several different power management policies could be contemplated, which differ with respect to the resulting power savings and the corresponding implementation complexity [4].

The simplest cluster level power management policy is to just allow each server to make its own voltage and frequency adjustments depending on the load submitted to it. All the optimizations of Section 6.5.1 could still be applied independently at each server. No further co-ordination is performed, and the load balancing switch just performs its normal function of approximately equalizing the load across all the servers. Although each node has the freedom to set its own operating conditions, we may find the settings to be roughly equal in practice, since the load balancer ensures approximately similar load for all the servers.

A more interesting power management policy at the server level is to combine the two policies in a co-ordinated manner:

1. switch servers off when the load is relatively low.
2. manipulate the voltage and frequency of each active server appropriately to deliver the required performance at the lowest voltage/frequency setting.

A simple model of the power consumption of the cluster of servers is necessary to work out the overall power management strategy. Let the dynamic power consumption of the CPU be given by:

$$\text{Dynamic CPU Power} = \frac{1}{2}\alpha C V^2 f \tag{6.8}$$

where α is the activity factor, V is the voltage, C is the effective capacitance, and f is the frequency of operation. The frequency also varies linearly with the voltage, making the power equation:

$$\text{Dynamic CPU Power} = c_1 f^3 \tag{6.9}$$

where c_1 is a constant.

We assume that the task latency is proportional to the frequency of operation. That is, if the frequency is doubled, then the task takes half the original time. This assumption is not quite accurate since the changed CPU frequency leads to differences in stall cycles, thereby affecting the total number of cycles in unpredictable ways, but is nevertheless useful for generating some important insights. The power consumed by the rest of the system (other components + leakage power in the CPU) is constant, so the total system power at frequency f can be expressed as:

$$P(f) = c_0 + c_1 f^3 \tag{6.10}$$

where c_0 and c_1 are constants.

For a cluster of n systems operating at frequency f_1, the total power is:

$$S\left(n, f_1\right) = n \times P\left(f_1\right) = n \times \left(c_0 + c_1 {f_1}^3\right) \tag{6.11}$$

If the frequency f_1 is too low the load balancer can turn off one server and re-distribute its load among the remaining $n-1$ servers. In this process, the power consumed by one server is subtracted from the total, but the re-distribution of the load increases the dynamic power of the remaining servers. If performance is not to be sacrificed, then the frequency of the remaining $n-1$ servers have to proportionately increase to $\frac{n}{n-1} f_1$. The power dissipated by the new cluster configuration is given by:

$$S\left(n-1, \frac{n}{n-1} f_1\right) = (n{-}1){\times}P\left(\frac{n}{n-1} f_1\right) = (n{-}1){\times} = \left(c_0 + c_1 \frac{n}{n-1} {f_1}^3\right) \tag{6.12}$$

The new configuration with $n-1$ servers dissipates less power than the n-server configuration when:

$$S\left(n-1, \frac{n}{n-1} f_1\right) < S(n, f_1) \tag{6.13}$$

From the expressions above, we can easily solve for f_1 and take the decision to turn a server off if the frequency of the n servers falls below f_1. Similarly, when f_1 is high, we can consider an additional server on, increasing the server count in the cluster to $n+1$. This permits us to deliver the same performance by reducing the server frequencies to $\frac{n}{n+1} f_1$). The addition is worth it when:

$$S(n, f_1) > S\left(n+1, \frac{n}{n+1} f_1\right) \tag{6.14}$$

From Equations 6.13 and 6.14, we can compute the optimal frequency ranges for an n-server cluster for different values of n. It should be pointed out that the above analysis is only an approximate indication of the actual relationships between power, performance, and frequency. In practice, this needs to be refined with other relevant information. For example, performance measured in terms of response time may not be exactly inversely proportional to frequency; all the required frequency points may not be available for a server; etc.

The approximate performance and power model used above can be replaced by a power management strategy using actual measurements of power and performance. Modern servers have sensors for reporting instantaneous power, which can be used along with performance information obtained from the operating system to build a fully dynamic control theory based strategy where decisions on frequency changes are taken in response to observed variations in power and performance as a consequence of current load and frequency/voltage settings [14].

Finally, we can drop the assumption we have made earlier about the cluster being homogeneous. In reality, large clusters centers are always heterogeneous simply because they are not fully populated at once – the specifications of later machines will most likely be different from the original set. The different machine types need to be first characterized for the variation of system power as a function of frequency and voltage settings, and also the performance metric at different operating condition settings [7,8]. In a scenario where a power constraint has to be met, the system can keep track of the projected power with the current cluster configuration and selection of operating conditions. If the power rises too high, then we evaluate the estimated performance loss by reducing the operating frequency at different nodes of the heterogeneous cluster, and select the server and its setting that leads to minimum performance loss while respecting the total power budget.

6.6 Summary

Power management at the server, cluster, and data center levels has emerged as an important problem in recent years, with energy costs of maintaining large computing infrastructure rivaling the cost of the systems themselves. Two major power management knobs available at this level of abstraction are: (i) dynamic voltage and frequency scaling for individual servers, and (ii) server consolidation, where we keep only as many servers in active state as necessary so that power can be saved by turning the rest off or setting them to low power mode. Simple techniques discussed in this chapter have been proposed as first attempts to manage power at these levels, but the topic continues to be an important research challenge. Accurate power and performance models are necessary for prediction of the impact of adjusting high level parameters. Server hardware continues to grow more sophisticated with the addition of sensors to report power, temperature, and other useful information that helps power management algorithms.

Server level power management strategies do have to be aware of long term consequences of the power optimizations performed, such as the impact of frequent on/off switching on disk reliability. In practice, the idealized situations discussed in this chapter regarding server task allocation needs to be refined. For example, all tasks may not be eligible to be executed on all servers because of restrictions arising out of client agreements [13]. A good data center level power management strategy of the future is likely to be co-designed with the mechanical analysis such as air-flow study.

References

1. Bohrer, P., Cohn, D., Elnozahy, E., Keller, T., Kistler, M., Lefurgy, C., Rajamony, R., Rawson, F., Hensbergen, E.V.: Energy conservation for servers. In: Proceedings of the IEEE Workshop on Power Management for Real-Time and Embedded Systems, pp. 1–4 (2001)
2. Chase, J.S., Doyle, R.P.: Energy management for server clusters. In: Proceedings of HotOS-VIII: 8th Workshop on Hot Topics in Operating Systems, May 20-23, 2001, Elmau/Oberbayern, Germany, p. 165 (2001)

3. Economou, D., Rivoire, S., Kozyrakis, C., Ranganathan, P.: Full-system power analysis and modeling for server environments. In: Workshop on Modeling Benchmarking and Simulation (MOBS) (2006)
4. Elnozahy, E.N., Kistler, M., Rajamony, R.: Energy-efficient server clusters. In: Power-Aware Computer Systems, Second International Workshop, PACS 2002 Cambridge, MA, USA, February 2, 2002, Revised Papers, pp. 179–196 (2002)
5. Fan, X., Weber, W.D., Barroso, L.A.: Power provisioning for a warehouse-sized computer. In: 34th International Symposium on Computer Architecture (ISCA), San Diego, California, USA, pp. 13–23 (2007)
6. Felter, W.M., Rajamani, K., Keller, T.W., Rusu, C.: A performance-conserving approach for reducing peak power consumption in server systems. In: Proceedings of the 19th Annual International Conference on Supercomputing, ICS 2005, Cambridge, Massachusetts, USA, June 20-22, 2005, pp. 293–302 (2005)
7. Heath, T., Diniz, B., Carrera, E.V., Jr., W.M., Bianchini, R.: Energy conservation in heterogeneous server clusters. In: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), Chicago, IL, USA, pp. 186–195 (2005)
8. Kotla, R., Ghiasi, S., Keller, T.W., III, F.L.R.: Scheduling processor voltage and frequency in server and cluster systems. In: High-Performance, Power-Aware Computing (HPPAC) (2005)
9. Lefurgy, C., Rajamani, K., III, F.L.R., Felter, W.M., Kistler, M., Keller, T.W.: Energy management for commercial servers. IEEE Computer **36**(12), 39–48 (2003)
10. Mansoor, A., Griffith, B.: Enabling high efficient power supplies for servers. Presentation at Intel Technology Symposium (2004)
11. Mitchell-Jackson, Jennifer, Koomey, J., Nordman, B., Blazek, M.: Data center power requirements: Measurements from silicon valley. Energy – The International Journal **28**(8), 837–850 (2003)
12. Miyoshi, A., Lefurgy, C., Hensbergen, E.V., Rajamony, R., Rajkumar, R.: Critical power slope: understanding the runtime effects of frequency scaling. In: Proceedings of the International Conference on Supercomputing, New York City, NY, USA, pp. 35–44 (2002)
13. Vasan, A., Sivasubramaniam, A., Shimpi, V., Sivabalan, T., Subbiah, R.: Worth their watts? an empirical study of datacenter servers. In: 16th International Conference on High-Performance Computer Architecture (HPCA-16), Raleigh, North Carolina, USA (2010)
14. Wang, X., Lefurgy, C., Ware, M.: Managing peak system-level power with feedback control. Tech. Rep. RC23835, IBM (2005)
15. Ware, M., Rajamani, K., Floyd, M., Brock, B., Rubio, J.C., Rawson, F., Carter, J.B.: Architecting for power management: The power7 approach. In: 16th International Conference on High-Performance Computer Architecture (HPCA-16), Raleigh, North Carolina, USA (2010)

Chapter 7
Low Power Graphics Processors

So far we studied power optimizations at various levels of design abstraction such as the circuit level, architectural level, all the way up to the server and data center level. In this chapter, we present a case study that combines several of the aforementioned techniques in a reasonably complex system: *a power efficient Graphics Processor*.

Computer graphics has progressed profoundly in recent years with applications in diverse fields. What started as a tool for military simulation ultimately evolved into an indispensable utility in a huge number of different application domains such as movie special effects, medical imaging, gaming, and computer-aided design. Since the applications are so varied, the kind of constraints they impose on graphics processing are also different. Applications such as games, animated chats, etc. are real time applications and need on-line synthesis of images. The emphasis is more on maintaining uniform frame rates, at times compromising the detail and precision at which the images are generated. On the other hand, graphics applications such CAD use images that are generated offline. The aim is to generate images at the highest level of detail without processing time constraints. Sometimes, huge workstations work for several hours to generate visually stunning images with special effects used in movies or to synthesize a view of an automobile part to the minutest detail used in CAD.

Advancements in semiconductor technology have made it possible to pack mobile devices with the computational capacity sufficient to port extremely complex applications on them. Some of the graphics applications such as screen savers, gaming, GPS-backed maps, animated chats, etc. that have been developed for either desktops or dedicated consoles are now emerging to be possible applications for mobile devices. The challenge in porting complex 3D graphics applications onto mobile platforms is posed not so much by performance as power consumption. Mobile devices are powered by battery, and since battery capacity is not increasing at par with processing power of chips, the gap between the demand and supply of power is widening. Increasing popularity of mobile graphics applications has introduced an additional dimension of power in the design of a graphics subsystem to the already existing ones of performance and quality. In recent years, industry and academia have been active in coming up with power optimizations for various aspects of graphics processing: (i) scaling down the applications to suit mobile platforms; (ii) low power graphics processor design; (iii) system level power reduction, etc.

P.R. Panda et al., *Power-efficient System Design*, DOI 10.1007/978-1-4419-6388-8_7,

Low power graphics system design forms the focus of this chapter. A detailed account of the operations in graphics processing is presented to provide a functional view of a graphics processor. This is followed by a discussion on the architecture of modern graphics processors highlighting the major power consuming components. Optimizations specific to each of these units and also the system level power management techniques are discussed further in the chapter.

7.1 Introduction to Graphics Processing

7.1.1 Graphics Pipeline

The aim of a graphics pipeline is to generate the view of a scene on a display device. The pipeline processes the complex geometry present in the scene, which is represented using several smaller primitives such as triangles, lines, etc., to produce the color corresponding to each position on a 2D screen called a *pixel*.

Several operations are applied in sequential order, on the data representing the mathematical model of an object, to create its graphical representation on the screen. The high level view of the flow of these operations, generally called the *Graphics Pipeline*, is illustrated in Fig. 7.1.

7.1.1.1 Application Stage

The application layer acts as an interface between the player and the game engine. Based on the inputs from the player, the application layer places the view camera, which defines the position of the "eye" of the viewer in 3D space. The application also maintains the geometry database, which is the repository of the 3D world and the objects used in the game, represented as geometric primitives (triangles, lines, points etc). Every object is associated with a *position* attribute defining its placement in the world space, and a *pose* defining the orientation of the object and movable-parts of the object with respect to a fixed point on the object as shown in Fig. 7.2.

Animation is the process of changing the position and pose of the objects from frame to frame, so as to cause the visual effect of motion. The movement of objects in a frame can be brought about by a combination of translation, rotation, scaling, and skewing operations (Fig. 7.3).

The application associates the objects in the geometry database with transformations as determined by the game play. The actual transformation operation on the primitives of the object happens in the next stage – the geometry stage of the pipeline. In addition, the application layer also identifies possible collisions among objects in the frame and generates a response in accordance with the game play. This layer is also responsible for processing the artificial intelligence (AI), physics, audio, networking, etc.

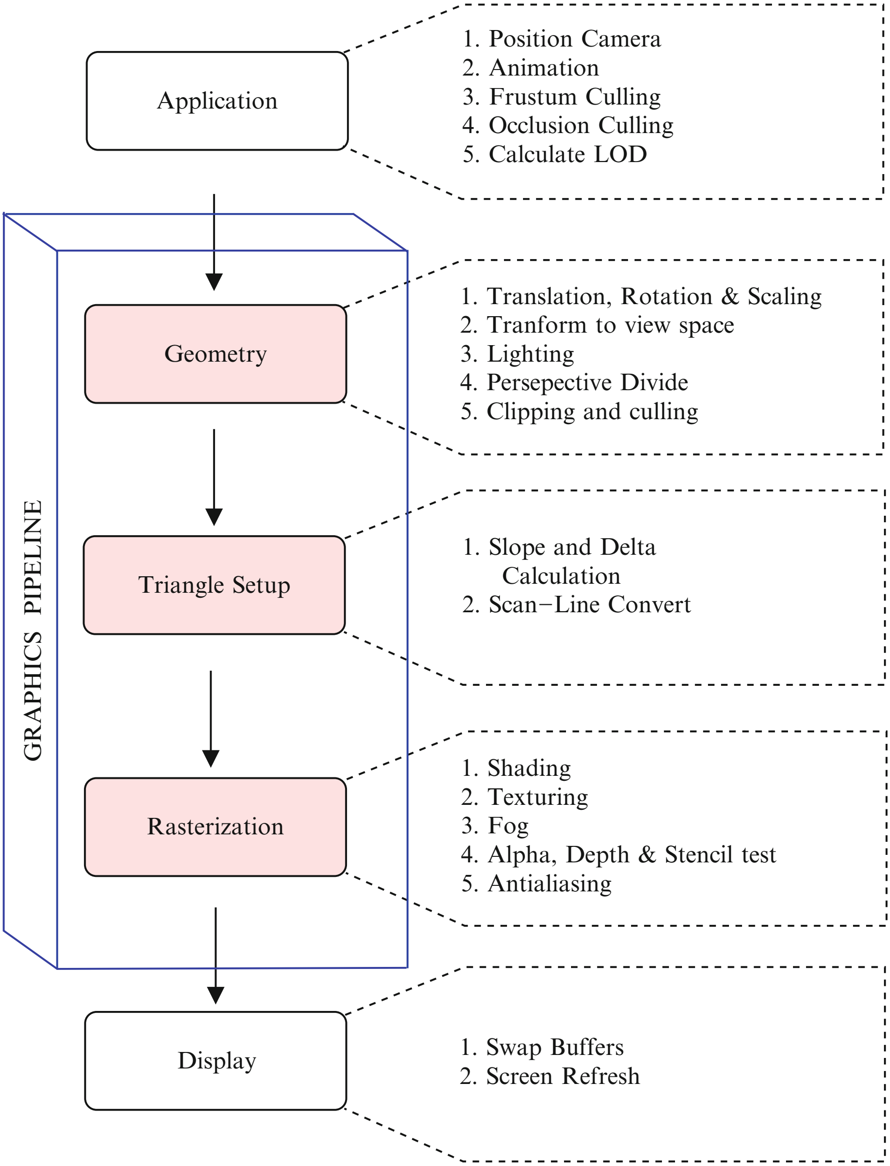


Fig. 7.1 Graphics Pipeline

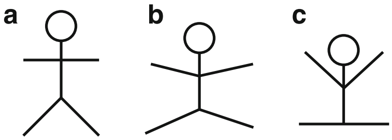


Fig. 7.2 (**a**), (**b**), and (**c**) Three different poses of the same character in a 3D game

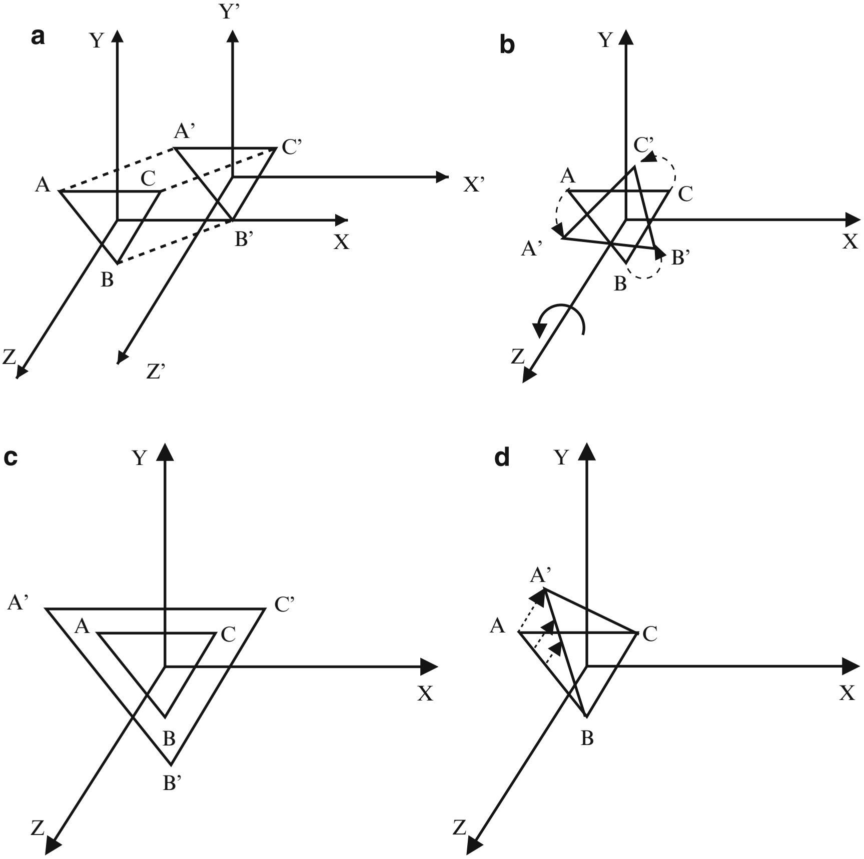


Fig. 7.3 Transformations on an object from ABC to A'B'C'. (**a**) Translation: displacement of an object from one position to another (**b**) Rotation: movement of an object around an axis causing angular displacement (**c**) Scaling: resizing of an object (**d**) Skewing: reshaping an object by scaling it along one or more axes

7.1.1.2 Geometry

The geometry engine receives the *vertices* representing the primitives of the objects as inputs from the application stage. The first step in the geometry stage is to apply the transformations (associated with them in the application stage) on the primitives. The various transformations are illustrated in Fig. 7.4. In newer pipeline implementations, the geometry engine is also capable of animating the primitives. In this case, the transformations are generated and applied by the geometry engine itself.

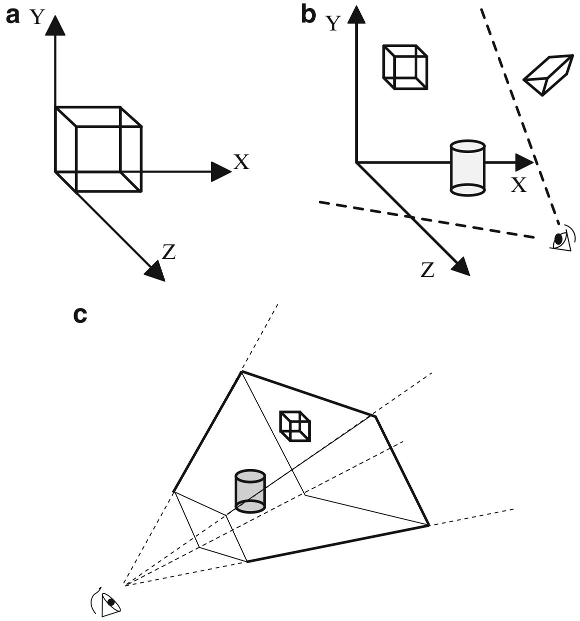


Fig. 7.4 Space-Space transformations on an object – (**a**) Model Space: centered at a point on the object (**b**) World Space: a common co-ordinate space us used for all the objects (**c**) View Space: the camera/eye forms the center of the world

In addition to these transformations, the geometry engine also needs to apply space-space transformations on the primitives. Various *spaces* used to represent a scene are illustrated in Fig. 7.4 and discussed below:

- **Model Space:** where each object is described with respect to a co-ordinate system centered at a point on the object.
- **World Space:** where all the objects that form the scene are placed in a common co-ordinate space.
- **View Space:** where the camera/eye forms the center of the world thus representing the world as seen by the viewer.

To transform the primitives from model space to view space, they are either first transformed to world space and then to view space, or directly transformed to view space. In terms of operations, these transformations are also a combination of translations and rotations. The next step is *lighting* the vertices taking into account the light sources present in the scene and also the reflections from the objects present in the scene. The lighting of primitives could be done either at vertex level or pixel level. Though pixel level shading results in better effects, the downside of per-pixel lighting is the resulting heavy computational workload. Thus, the choice between per-vertex and per-pixel shading is a trade-off between accuracy and workload.

After the per-vertex operations of transformation and lighting are done, the vertices are assembled into triangles. Before the triangles are sent to the next stages of the pipeline for further processing, the primitives that would not contribute to the pixels that are finally displayed on the screen, are discarded so as to reduce the workload on the pixel processor. As a first step, the geometry engine identifies the triangles that fall partially or totally outside the view frustum. The primitives that fall totally outside the frustum are *trivially rejected*. If the primitives are not completely outside the frustum, they are divided into smaller primitives so that the part falling outside the frustum can be *clipped off*. In addition to the primitives falling totally outside the view frustum, the triangles that face away from the camera are also trivially rejected. This process is called *back-face culling*. For example, independent of the viewing point, half of a solid sphere's surface is always invisible and hence can be discarded.

7.1.1.3 Triangle Setup

So far in the pipeline, the scene is represented in terms of triangles, lines, and points. But what is finally displayed on the screen is a 2D array of points called *pixels*. In order to progress towards this final objective, the triangles are first divided into a set of parallel horizontal lines called *scan-lines*, as shown in Fig. 7.5. These lines are further divided into points, thus forming the 2D array of points called the *fragments*. The scan-line conversion of the triangles occurs during the triangle setup phase; the scan lines are then passed on to the rasterization engine, which then generates pixels from these lines. While dividing the triangles into the corresponding scan lines, the triangle setup unit calculates the attributes – depth, color, lighting factor, texture coordinates, normals, etc., of the end points of the lines through interpolation of the vertex attributes of the triangle.

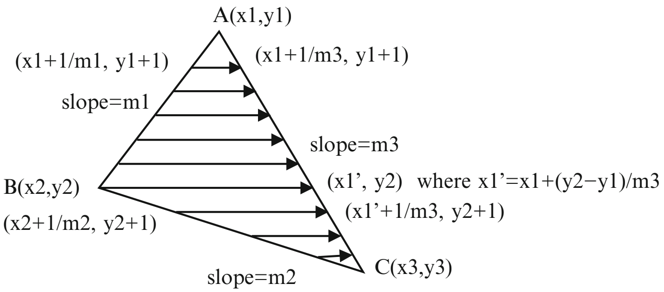


Fig. 7.5 Scan line conversion: the process of dividing the primitives into parallel scan-lines

7.1.1.4 Rasterization

The raster engine generates the pixels from the scan-lines received from the setup unit. Each pixel is associated with a *color* stored in a *color buffer* and *depth* stored in a *depth buffer*. These two buffers together form the *framebuffer* of the graphics processor. The aim of the pixel processor is to compute the color of the pixel displayed on the screen. The various operations involved in this processing are enumerated below:

Shading: In this step, the lighting values of the pixels are computed. This is done by either assigning the weighted average of vertex lighting values or, for greater accuracy, actually computing the lighting at each pixel.

Texturing: *Texture mapping* is a technique of adding surface detail, texture, or color to an object and helps significantly in adding realism to the scene. The process of texture mapping can be explained with a simple example shown in Fig. 7.6. Consider modeling a globe. One way to do this is to represent the sphere as a large number of tiny triangles and associate the vertices of these triangles with appropriate colors so that after the triangles are passed through the pipeline, what finally appears on the screen looks like a globe. The modeling effort in this case is so huge that it makes rendering such models almost impossible. Things would be easier if we could just define the mapping of a few points on a sphere to the points on a 2-D world map, and the pipeline had the capability to associate the pixels with appropriate colors from the world map. This process of mapping the pixels on a 3D model to points on a 2D texture called *texels*, is called texture mapping or *texturing*. This has made the generation of objects with surface irregularities such as the bumps on the surface of moon, objects with surface texture such as that on a wooden plank, etc., possible in a graphics pipeline.

Fog: After texturing, *fog* is added to the scene, giving the viewer perception of depth. Fogging effect is simulated by increasing the haziness of the objects with increasing distance from the camera. The fog factor is thus a function of the z-value of a pixel and could increase either linearly or exponentially. The fog factor is then applied to the pixel by blending it with the color computed in shading and

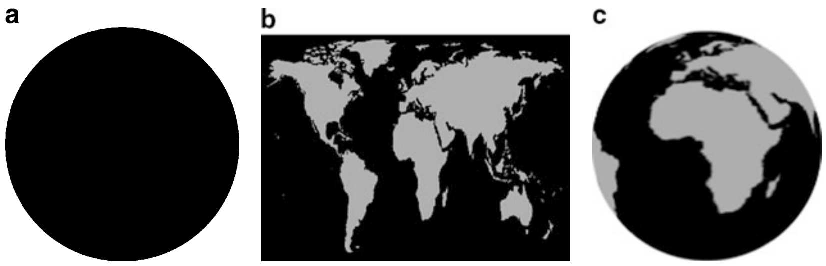


Fig. 7.6 Texture mapping example. (**a**) Object. (**b**) Texture. (**c**) Textured object

texturing steps. Another application of fogging is to make the clipping of objects at the far clipping plane less obvious by fading their disappearance rather than abruptly cutting them out of the scene.

Alpha and Depth: Alpha value is one of the attributes of a vertex that is used to model opacity of the vertex. This is required to model transparency and translucency of the objects, for example in simulating water, lens, etc. An opaque object occludes the objects that are behind it. Thus, if a pixel is opaque and the z-value of the pixel is less than the value present in the depth buffer at the position corresponding to the pixel, then the depth buffer and color buffer are updated with the attributes of the pixel. However, if the object transparent, depending on its transparency, the color of the occluded objects have to be blended with the color of the object to simulate the effect of transparency. In addition to depth and color buffers, a graphics pipeline also has a *stencil buffer*. Generally, this buffer stores a value of 0/1 per pixel to indicate whether the pixel has to be masked or not. This is used to create many effects such as shadowing, highlighting, and outline drawing. The operations involved in these three tests together can be summarized as follows (Algorithm 1).

Algorithm 1 Alpha, depth and stencil test

```
if StencilBuffer (x,y) $\neq$ 0 then
    if Alpha $\neq$ 0 then
        if DepthBuffer (x,y) $\geq$ z then
            ColorBuffer (x,y) $\leftarrow$ color
            DepthBuffer (x,y) $\leftarrow$ z
        end if
    end if
end if
```

Anti-aliasing: When an oblique line is rendered, it appears jagged on the screen as shown in Fig. 7.7(a). This is a result of discretization of a continuous function (line) by sampling it over a discrete space (screen). One way to alleviate this effect is to render the image at a resolution higher than the required resolution and then filter down to the screen resolution. This technique is called *full screen anti-aliasing* (Fig. 7.7(a)). The problem with this method is that it increases the load due to pixel

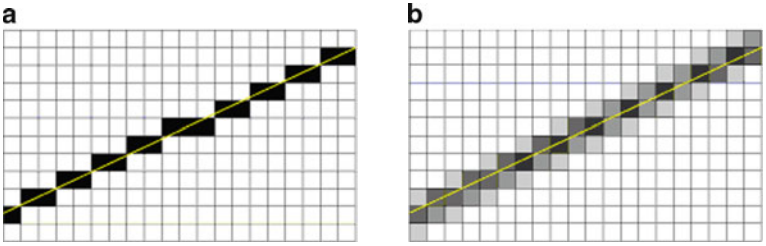


Fig. 7.7 Anti-aliasing illustration. (**a**) Line. (**b**) Anti-aliased line

processing. Hence an optimization called *multi-sampling* is generally used, which identifies the edges of the objects in the screen and applies anti-aliasing only to the edges.

7.1.1.5 Display

When a new frame is to be displayed, the screen is first cleared; then the driver reads the new frame from the framebuffer and prints it on the screen. Generally a screen refresh rate of 60fps is targeted. If only one buffer is used for writing (by the GPU) and reading (by the display driver), artifacts such as flickering are common because the GPU could update the contents of the frame before they are displayed on the screen. To overcome this problem, generally *double buffering* is used, as shown in Fig. 7.8, wherein the display driver reads the fully processed frame from the front buffer while the GPU writes the next frame to the back buffer. The front and back buffers are swapped once the read and write operations to the front and back buffers respectively, are completed. The obvious drawback of double buffering is performance loss. Since the frame which needs slightly more than 16.67msec (1/60 of a sec) would be updated only in the next refresh cycle, the GPU cannot start processing the next frame until then. In such cases, the overall frame rate may fall to half the targeted frame rate even when the load is only slightly increased. To counter this problem, *triple buffering* using three buffers (one front and two back buffers) can be used. The GPU can now write to the additional buffer, while the

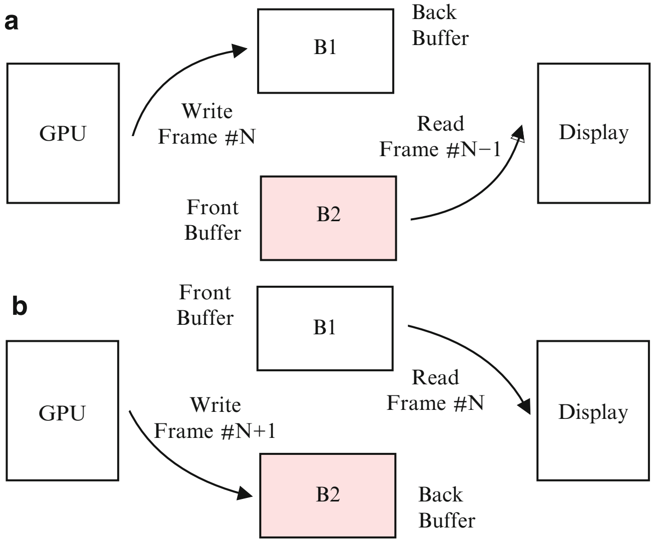


Fig. 7.8 Double Buffering. (**a**) GPU writes to B1 and Display driver reads from B2. (**b**) Swap Buffers: GPU writes to B2 and driver reads from B1

other buffer holds the frame to be refreshed. The choice of double buffering or triple buffering depends on the availability of memory space.

7.1.2 Graphics Processor Architecture

The computational workload of 3D graphics applications is so high that to achieve real time rendering rates, hardware acceleration for graphics processing is almost always necessary. Generally, the application layer executes on the CPU and the rest of the graphics processing is offloaded to Graphics Processing Units (GPUs). To enable ease of development and also application portability, an Application Programming Interface (API) is used to abstract the hardware from the application. The device driver that forms the interface between the CPU and GPU receives the API calls from the application and interprets them to the GPU. The interaction between CPU and GPU is shown in Fig. 7.9.

The commands from the application running on the CPU are passed on to the GPU through a ring buffer interface. The data associated with these commands, such as vertex attributes, textures, and shader programs are transferred from system memory to *VRAM* through Direct Memory Access (DMA) transfers. In addition to acting as temporary storage for input data, VRAM also needs to store the pro-

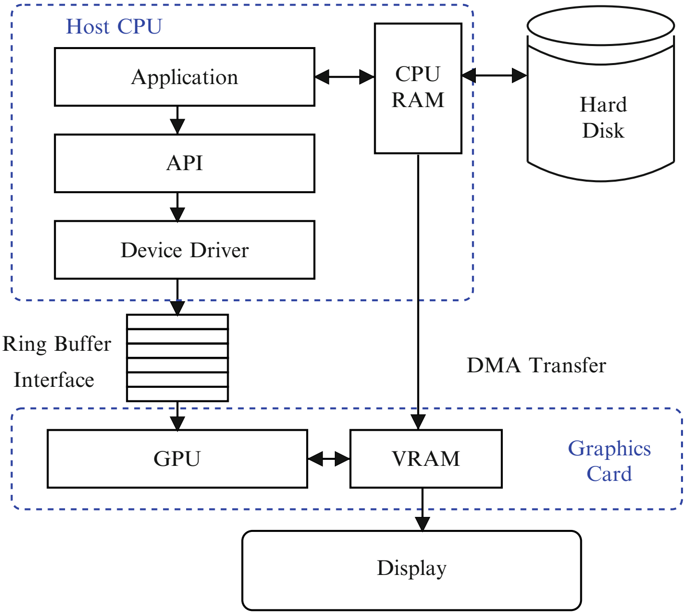


Fig. 7.9 Interaction between CPU and GPU: Commands are transferred through the Ring buffer interface and the data through DMA transfers

cessed frames that are ready for display. This area in VRAM that is reserved for storing the processed frames is essentially the framebuffer. Since the GPU need not send the processed frames to CPU, the CPU need not wait for the GPU to complete the processing before issuing the next GPU command. This helps CPU and GPU to work in parallel, thus increasing the processing speed.

In graphics applications, we observe that the input data set is operated upon by a large number of sequential operations. Hence, GPUs are generally deeply pipelined to enhance the throughput. Moreover, the data set is huge and operations on one data element are independent of operations on other data elements. Hence, each stage in the pipeline consists of multiple function units to support parallel processing of the data streaming into it. The Fig. 7.10 shows the high level architectural view of a graphics processor.

The *Host Interface* acts as an interface between the host CPU and the GPU pipeline. It maintains the state of the pipeline, and on receiving the commands from the driver, it updates the state and issues appropriate control signals to the other units in the pipeline. It also initiates the required DMA transfers from system memory to GPU memory to fill *vertex buffer* and *index buffer*, load the shader program, load textures, etc. Vertex buffer is generally implemented as a cache, since re-use of vertices is expected.

The first block in the pipeline, *Transformation and Lighting*, is responsible for performing the transformation and lighting computations on the vertices. The vertex input cache and index buffer are used to buffer the inputs to this block. The primitives of an object are generally found to share vertices as shown in Fig. 7.11 – vertex 3 is common to triangles $T1, T2$, and $T3$. *Index mode* for addressing the vertices results in reduced CPU-GPU transfer bandwidth than transferring the vertices in the presence of vertex reuse [13].

For the example shown in Fig. 7.11, if we send the vertices forming each of the triangles T1, T2, and T3 to the GPU, we need to send 9 vertices as shown in Fig. 7.12. If each vertex is made of N attributes, where each attribute is a four component vector (e.g., x,y,z,w components for position; R,G,B,A components of color), a bandwidth of $9 \times 4 \times N$ floating point data is required. Instead, we could assign each vertex an index (a pointer to the vertex – of integer data type), and send 9 indices and only 5 vertices to the GPU. Thus, in indexed mode we send only 9 integers and $5 \times 4 \times N$ floating point data. Indexed mode for vertex transfer and the resulting bandwidth saving are depicted in Fig. 7.12. The indices of the vertices to be processed are buffered into the index buffer and the attributes of these vertices are fetched into the vertex input cache, since they are expected to be reused [5]. The processed vertices are also cached into a *vertex output cache* so as to reuse the processed vertices. Before processing a new vertex, it is first looked up in the vertex output cache [5]. If the result is a *hit*, the processed vertex can be fetched from the cache and sent down the pipeline, thereby avoiding the processing cost.

The transformed and lit vertices are sent to the *Primitive Assembly* unit which assembles them into triangles. These triangles are sent to the *Clipper* where trivial rejection, back-face culling, and clipping take place. The triangles are then sent to the Triangle Setup unit which generates fragments from the triangles. The scan-line

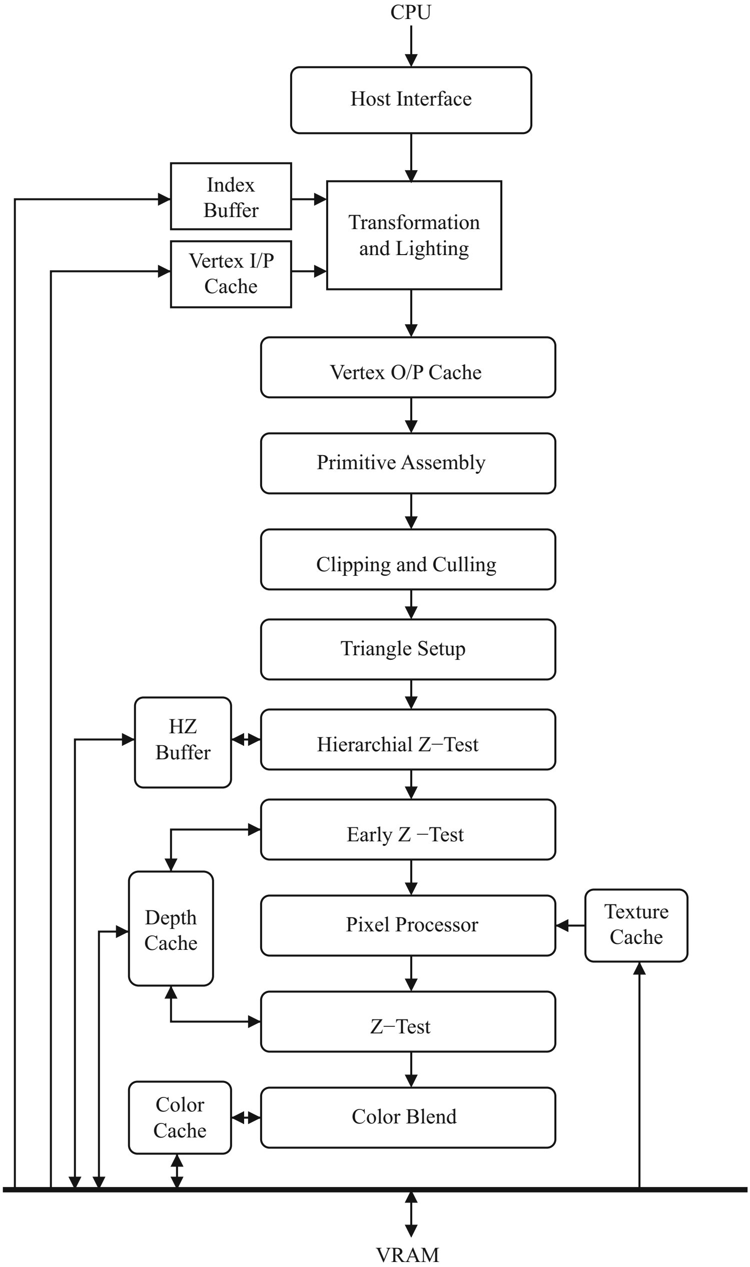


Fig. 7.10 Fixed function Graphics Processor Architecture

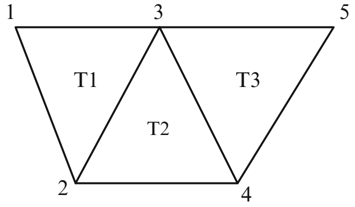


Fig. 7.11 Triangles sharing vertices

b

Vertex	v1	v2	v3	v4	v5
Index	1	2	3	4	5

a

T1	v1	v2	v3
T2	v2	v3	v4
T3	v3	v4	v5

T1	1	2	3
T2	2	3	4
T3	3	4	5

Fig. 7.12 Indexed addressing into vertex buffer needs a transfer of 5 vertices and 9 indices instead of 9 vertices. (**a**) Triangles represented in terms of vertices. (**b**) Indexed triangle representation

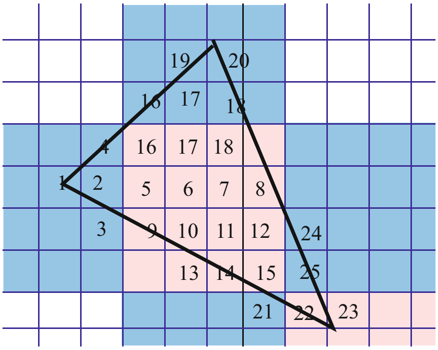


Fig. 7.13 Tiled Triangle Traversal: All pixels belonging to a tile are generated before moving on to the pixels falling in the next tile. Spatial locality into Framebuffer and Texture cache is hence exploited to reduce memory bandwidth requirement

conversion of triangles into lines does not exploit the spatial locality of accesses into the framebuffer and also the texture memory. Hence tiled rasterization, as shown in Fig. 7.13, is generally employed. In this technique, the screen is divided into rectangular tiles and triangles are fragmented such that the pixels belonging to same tile are generated first before proceeding to pixels falling in a different tile. The accesses to the framebuffer and texture cache are also matched to this tile size so that accesses to memories can be localized [16].

The next unit is the *pixel processor* which shades and textures the pixels. Since texture accesses exhibit high spatial and temporal reuse, a dedicated cache called the *Texture Cache* is used in this unit to cache the textures. Most architectures use depth based optimizations prior to pixel processing because a large number of fragments are often culled in the depth test that follows pixel processing. Thus, the time spent on shading and texturing such fragments is actually wasted. However, it is not possible to conduct the depth test prior the pixel processing because, the pixel processor can potentially change the depth or transparency of the pixel. In circumstances where it is known that the pixel processor would not change these parameters, we can always perform the depth test prior to pixel processing. This is known as the *earlyZ* test [19]. It is generally observed that if a pixel fails the depth test, the pixels neighboring it also fail the depth test with a high probability. This property is exploited by the *Hierarchical Z Buffer* algorithm in identifying the groups of pixels that could be culled, thus reducing the number of per-pixel z-tests [3, 7].

After being shaded and textured, the pixels are sent to the *Render Output Processor* (ROP) for depth, stencil, and alpha tests followed by blending, and finally, writing to the framebuffer. Generally, the z-cache and color cache are used in this block to exploit spatial locality in the accesses to the off-chip framebuffer.

The initial generations of GPUs were completely hardwired. However, with rapid advances in computer graphics, there is a need to support a large number of newer operations on vertices and pixels. Fixed function implementations have been found inadequate to support the evolving features in the field of graphics processing due to their restricted vertex and pixel processing capabilities. Programmable units to handle vertex and pixel processing have been introduced in the *programmable graphics processors* of recent years. The vertex and pixel programs that run on these programmable units are called *Shaders*. By changing the shader code, we can now generate various effects on the same graphics processing unit.

A study of the workload characteristics of various applications on modern programmable processors reveals that the relative load due to vertex processing and pixel processing varies with applications and also within an application [2]. This results in durations when the vertex processors are overloaded while the pixel processors are idle and vice-versa, leading to inefficient usage of resources. The resource utilization efficiency can be improved by balancing the load for both vertex and pixel processing on the same set of programmable units, leading to faster overall processing. This is illustrated in Fig. 7.14).

Modern games are expected to have a primitive count of about a million resulting in tens of millions of pixels. The operations on these millions of vertices and pixels offer the scope for a very high degree of parallelism. Moreover, large batches of these vertices and pixels share the same vertex shader and pixel shader programs respectively. Hence, the programmable units are generally designed as wide SIMT (Single Instruction Multiple Thread) processors. In Fig. 7.14, we observe that the GPU consists of multiple programmable units, each consisting of several processing elements (PEs). Different threads could be running on different programmable units, but within a programmable unit, the same thread is executed on a different data

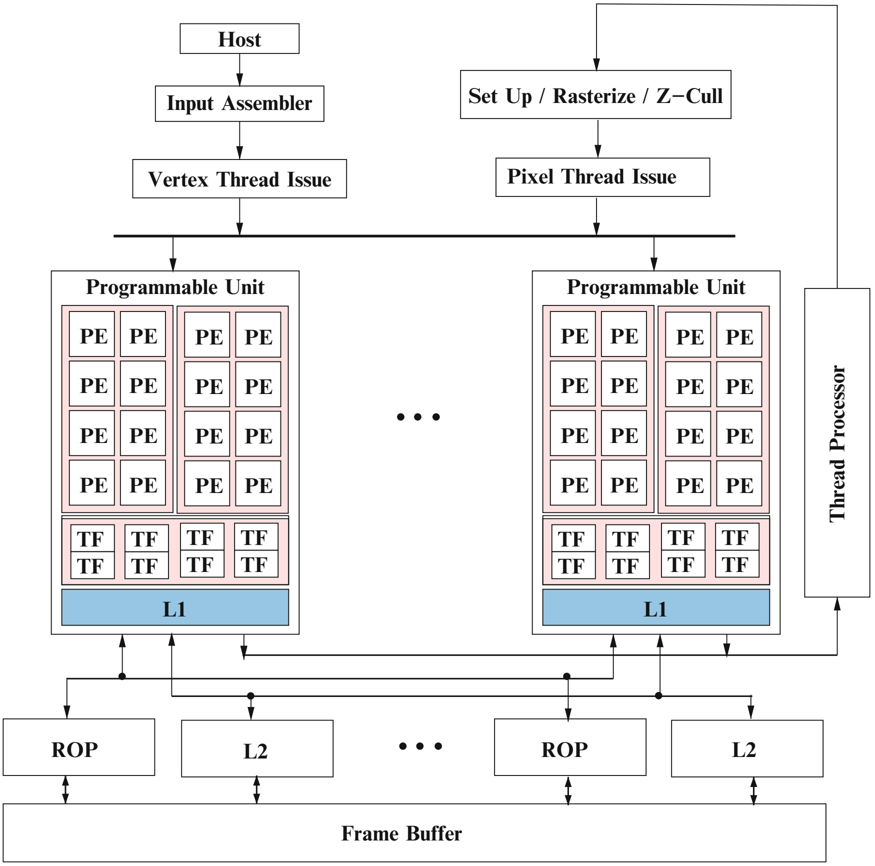


Fig. 7.14 Unified Shader Architecture for Graphics Processor

element in every PE. All these PEs can hence share the same instruction memory and decoder. This results not only in optimization of area, but also in considerable power savings because the costs of instruction fetch and decode are amortized over the group of threads running in tandem on the PEs of the programmable unit.

7.1.3 Power Dissipation in a Graphics Processor

From the analysis of operations in a graphics pipeline, it is observed that most of the computations are concentrated in the programmable units, texture units, and ROP units. Programmable units execute a large number of floating point vector operations; texture units use large memory bandwidth to move the textures from VRAM to cache, and perform a large number of floating point operations for

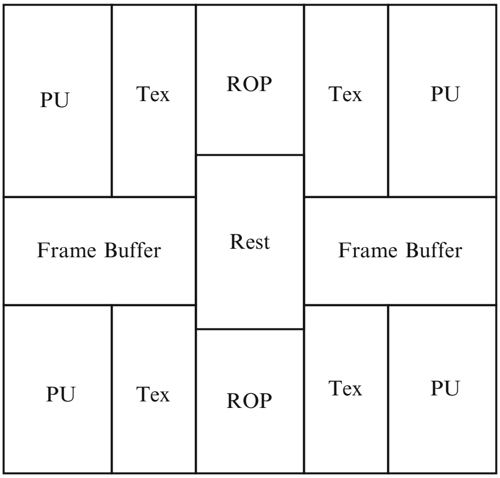


Fig. 7.15 Footprint of the die of GT200 {approximately to the scale} [1]: PUs, ROPs and Texture units occupy most of the chip's real estate

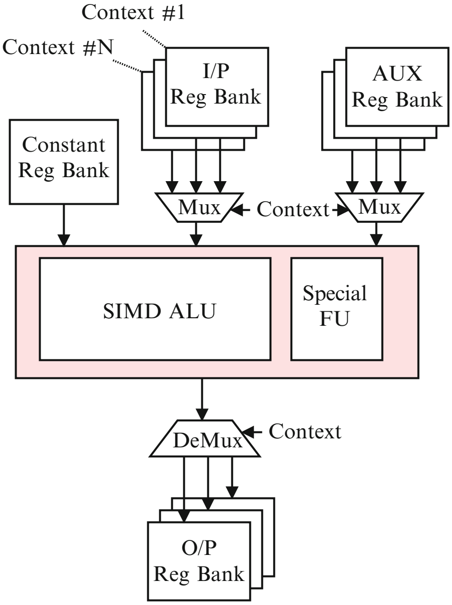
filtering the texels; ROP units are memory intensive, needing multiple reads and writes to the color and depth buffers. This is also obvious from the footprint of the die of nVidia's graphics processor (GT200 targeted for laptop computers) shown in Fig. 7.15; most of the die area is occupied by these three units.

Since the three units identified above occupy most of the real estate on the die and also contribute to a major fraction of the power consumption, most of the attention in low power graphics also revolves around these three units. Optimizations targeting each of these units are discussed in detail in the following sections.

7.2 Programmable Units

The high level view of a processing element(PE) in the Programmable Unit is shown in Fig. 7.16. Each processing element in the programmable unit consists of a SIMD ALU working on floating point vectors. In addition to the SIMD ALU, there is a also a scalar ALU that implements special functions such as logarithmic, trigonometric, etc. The ALU supports multi-threading so as to hide the texture cache miss latency. Context switches between threads in a conventional processor causes some overhead since the current state (consisting of inputs and the auxiliaries generated) needs to be stored in memory and the state of the next thread has to be loaded from memory. In order to support seamless context switches between the threads, the PUs in a graphics processor store the thread state in registers. The register file in the shader has four banks, one each to store input attributes, output attributes, constants, and intermediate results of the program. The *constant register bank* is shared by all threads whereas separate input/output and temporary registers are allocated to each thread. The instruction memory is implemented either as a scratch pad memory where the driver assumes the responsibility of code transfer or through a regular cache hierarchy.

Fig. 7.16 Processing Element (PE)



7.2.1 *Clock Gating*

Clock gating of the various sub-blocks of a programmable unit presents itself as a huge power saving opportunity. Since the PUs support a large number of threads and use registers to save the state, large register files are needed. However, since only one thread is active at any given instant, it is sufficient to clock only the registers allotted to the active thread and gate the clock to the remaining the registers. Similarly, the special function units in the ALU are infrequently used, and hence, can be activated only when the decoder stage confirms the need.

7.2.2 *Predictive Shutdown*

Predictive shutdown is an effective technique for reducing power loss due to leakage in the idle components of a system. Due to workload variations, not all programmable units are fully utilized in every frame. Advance information about a frame's workload can help estimate the number of cores required to process it within the time budget. By activating only the required number of cores and powering down the surplus ones, leakage power from the idle cores can be avoided, thereby leading to substantial power savings. A history based method could be used to estimate the utilization of the PUs [24]. Let the number of active cores used to process the n^{th} frame be S_n and the rate at which it was processed be FPS_n. Then the maximum rate at which each of the cores processed the frame is given as $\frac{FPS_n}{S_n}$. Similarly, the maximum number of cores required to process $n + 1^{th}$ frame can be caculated as

$$S_{n+1} = \frac{\text{Target frame rate for } n + 1^{th} \text{ frame}}{\text{minimum rate at which a core is estimated to process the frame}} \qquad (7.1)$$

The expected rate at which the core processes a frame can be approximated to the mimimum processing rate observed in processing of a window of m previous frames.

Based on previous history, the number of active cores S_{n+1} required to process the $n + 1^{th}$ frame is given by the equation:

$$S_{n+1} = \frac{[FPS_{target} + \alpha]}{min\{\frac{FPS_n}{S_n}, \frac{FPS_{n-1}}{S_{n-1}}, \ldots\ldots, \frac{FPS_{n-m+1}}{S_{n-m+1}}\}} \qquad (7.2)$$

The factor α is introduced so as to slightly overestimate the core requirement, so that small variations in the workload can be taken care of without missing the deadline. In the above formula, it is assumed that the entire duration in processing the frame is spent on the PUs, which is generally not true. The frame could as well be texture intensive or ROP intensive. The α factor also serves to reduce the effect of deadline misses due to under estimation of workload.

7.2.3 Code Transformation

Several scalar and vector code transformations and compiler optimizations applicable to generic SIMD code are also be applicable to the vertex and pixel shaders. In addition, there are transformations that are specific to graphics processing. One such transformation that divides the vertex shader into multiple passes to save power due to geometry processing is described in this section [21].

Geometry – the measure of the number of objects present in the scene and the level of detail at which they are modeled, is one of the most important aspects determining the complexity and visual reality of a scene. The increasing emphasis on the incorporation of intricate details in a scene is leading to an increase in the number of primitives per frame, since modeling at finer levels of granularity requires the objects to be represented with a large number of smaller primitives. As a result the workload due to geometry processing in the modern games is significantly impacting the performance and also power consumption of modern games.

It has been observed from the simulation of games and benchmarks, that on an average about 50% of primitives are trivially rejected in each frame. Trivial rejects comprise the primitives that fall totally outside the viewing frustum and also front/back face culled primitives. Since testing for trivial rejection requires

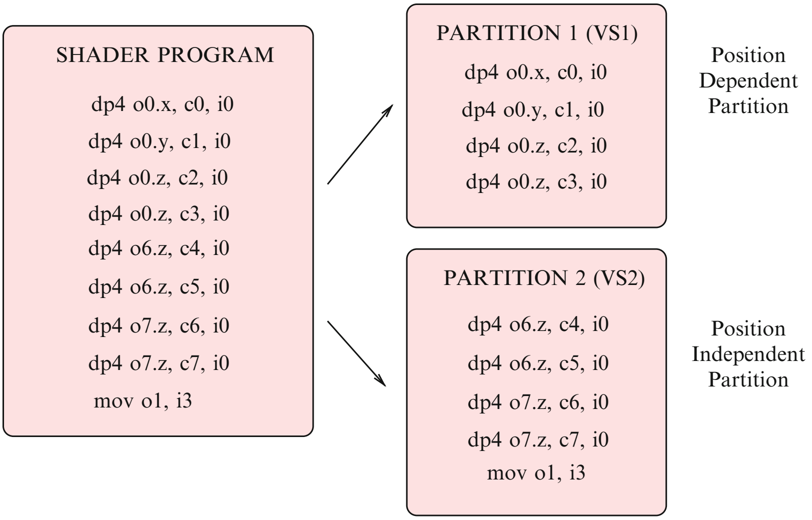


Fig. 7.17 Vertex Shader partitioning into position variant (VS1) and position invariant (VS2) shaders. In this example, o0 is the output position attribute

only the position information of the vertex, the time spent on processing the non-transformation part of the vertex shader on these vertices is wasteful. Instead, the vertex shader can be partitioned into position variant (transformation) and position invariant (lighting and texture mapping) parts. The position invariant part is deferred until after the post trivial reject stage of the pipeline, achieving significant savings in cycles and energy expended on processing these rejected vertices. An example illustrating vertex shader partitioning is shown in Fig. 7.17.

The changes to be incorporated in the conventional graphics pipeline to introduce partitioned vertex shading are shown in Fig. 7.18. In the modified pipeline, the VS1 stage computes only the position variant part of the vertex shader and the rest of the vertex processing is deferred to the VS2 stage. The Clipper stage is divided into *Trivial Reject* and *Must Clip* stages. Triangles passing through the trivial reject test are sent to the VS2 stage after disassembling them into vertices (since the vertex shader can only work on vertex granularity). These vertices, after being processed in VS2, are assembled back to triangles and sent to the Must Clip stage. The geometry engine of the pipeline is thus modified and the fragment generation and rendering takes place as it was in the original pipeline.

From the discussion so far it might appear that it is most appropriate for the API to support vertex shader partitioning. This would require the application developer to provide two pieces of vertex shader programs – one for transforming the vertices and one for lighting and texturing. But such hard partitioning of shaders is not always viable due to the following reasons.

- There could be a significant number of instructions common to position-variant and position-invariant parts of the vertex shader. This leads to code duplication.

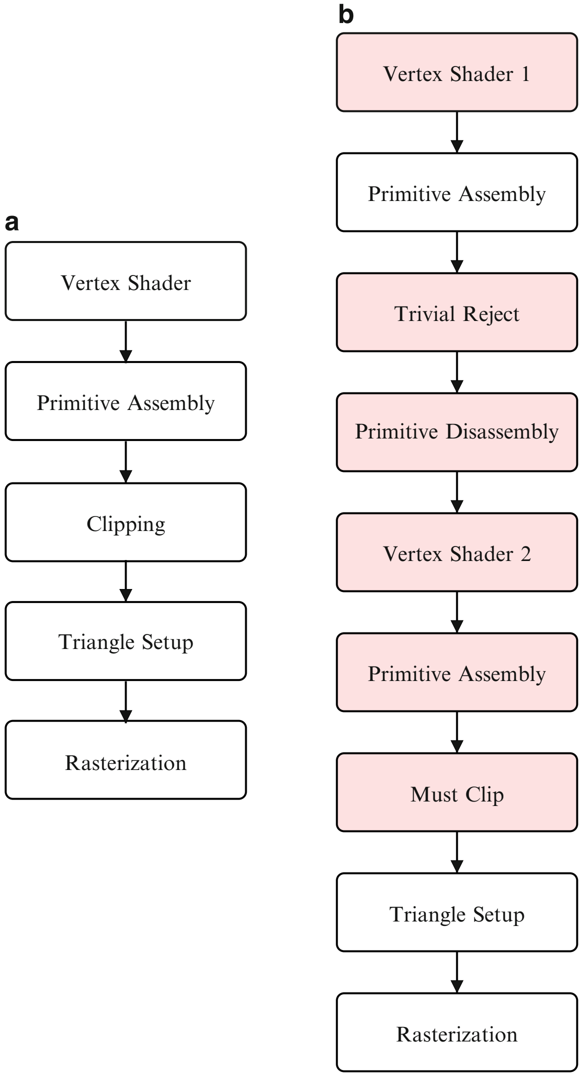
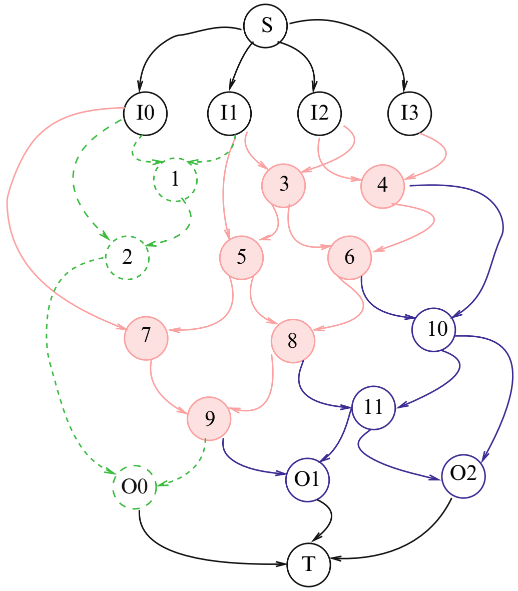


Fig. 7.18 Pipeline Modified to support Vertex Shader Partitioning. (**a**) Conventional Pipeline. (**b**) Modified Pipeline

- Thread setup overhead for the second vertex shader could overshadow the advantage of deferring the position invariant part of the shader code. Hence, an adaptive algorithm for vertex shader partitioning could be used, where the partitioning decision is based on a trade-off between the setup overhead and cycles saved due to partitioning.

Fig. 7.19 DAG of a vertex shader. Nodes 1,2 contribute only to position variant part of vertex shader. Nodes 10,11 contribute to position invariant partition. Nodes 3-9 are shared by both position variant and invariant parts of the vertex shader



To tackle the issue of code duplication, some of the auxiliary variables that are generated in stage VS1 and re-used in stage VS2 could be transmitted to stage VS2. For example, in the DAG shown in Fig. 7.19 where each node represents the operands of the instructions of the shader, the solid nodes are shared by both position variant and position invariant parts of the shader.

If these common auxillary variables can be passed on to VS2, cost of recomputing them in VS2 can be avoided. The number of these variables is limited by the fact that most of the shader architectures have a limit on the number of inputs to the programmable units. But since most of the time, the shaders do not need as many inputs, some of the auxiliary variables also can be sent as inputs to stage VS2. Data flow analysis of the vertex shader would lead to identification of best set of auxiliary variables to be transmitted from VS1 to VS2 so that the cost due to code duplication is minimized.

Spawning a thread on the Programmable Shader Unit incurs some thread setup overhead, the extent of which is dependent on the micro-architecture of the thread setup unit. This could include the idle time waiting for the availability of resources, time spent on loading the inputs, time spent on transmission of outputs, etc. Vertex shader partitioning incurs an overhead for setting up VS2 threads. Hence it is important to weigh the benefit of cycles saved on rejected vertices against the overhead incurred on thread setup for the vertices that are not rejected.

Let ST and ET be the setup overhead and execution time of the shader without partitioning; ST_1 and ST_2 be the setup overheads, and ET_1 and ET_2 be the execution times of VS_1 and VS_2 respectively. The cost of processing a batch of vertices (B) without vertex shader partitioning is given as

$$Cost_{no-part} = B \times (ST + ET) \tag{7.3}$$

If C is the rate at which vertices are trivially rejected, then the cost incurred to process the batch with partitioning is given by:

$$Cost_{part} = B \times (ST_1 + ET_1) + B \times (1 - C) \times (ST_2 + ET_2) \tag{7.4}$$

Vertex shader partitioning is profitable only if $Cost_{part}$ is less than $Cost_{no-part}$. Since the clip rate is known only after processing the frame, it is not possible to use the above cost functions to take the partitioning decision prior to processing the frame. However, due to spatial coherence of the frames, the trivial reject rate of adjacent frames is observed to be comparable. Thus, the clip rate of the previous frame can be taken as an approximation for the clip rate of present frame and the driver can be enhanced to take the partitioning decision on-the-fly based on the clip rate history and overheads and execution times of VS1 and VS2 shaders.

Thus, the history based adaptive vertex shader partitioning can be used to avoid redundant operations in the pipeline, thereby saving power.

7.3 Texture Unit

Texture mapping is the process of mapping an image in texture space on to a surface in object space using some mapping function [18]. Since texture space and object space could be at arbitrary distance and orientation with respect to each other, there is no one-to-one correspondence between the pixels on the object and texels of the texture. This necessitates the use of some texture filtering mechanism to attribute the best color to a pixel.

Bilinear filtering and *Trilinear filtering* [18] are two most common filtering techniques used for texture mapping. In bilinear filtering the weighted average of four texels nearest to the pixel center gives the color of the pixel (Fig. 7.21). In order to produce good results for various levels of depth (*lod*) at which the object could be viewed, the texture image is stored at various resolutions called *mipmaps* [18]. The nearest mipmap is picked for filtering at run time based on the *lod*. In trilinear interpolation, the bi-linearly interpolated values from the two nearest mipmap levels are averaged to give the color of the pixel.

Standard cache based memory architecture could be used for the texture memory accesses since it exploits locality of accesses. However, this is still expensive in terms of power, as each access results in a cache lookup operation with the standard power overheads. A custom memory architecture that uses some knowledge of texture access patterns can save power over a conventional cache architecture.

A second major source for power dissipation in a texture unit is the power expended in fetching the textures from off-chip VRAM to the on-chip texture cache. We also discuss various texture compression techniques that ameliorate this problem.

7.3.1 Custom Memory Architecture – Texture Filter Memory

Since the direction of accesses in texture memory is arbitrary, a blocked representation of texture maps in memory is generally used. This is illustrated in Fig. 7.20. Texels within a block reside in contiguous memory space, similar to the tiled storage in Section 4.7.2. The computation of the texel address from the texel co-ordinates is shown in Algorithm 2. The overhead of extra additions and shifts in the block address computation is offset by the performance gained by the reduced cache miss rates by selecting the line size equal to the block size [11].

Texture mapping with bilinear filtering exhibits high spatial and temporal locality. This is because:

- to compute the color of a pixel we need to fetch four neighboring texels,
- consecutive pixels on the scan line map to neighboring texels, and
- consecutive scan-lines of a primitive share texels.

In addition to locality, texture mapping also exhibits predictability in access pattern. As seen in Fig. 7.21, access to texel $t1$ is followed by accesses to texels $t2$, $t3$, and $t4$. Thus, the $t1$ access gives us advance information about the future texel accesses. However, all the other three texels might not lie in the same block. The different cases that arise are illustrated in Fig. 7.22. The information about which of

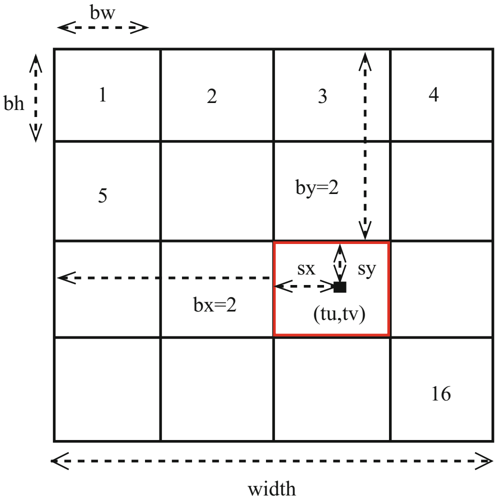


Fig. 7.20 Blocked representation of texture

Algorithm 2 Computation of texel address

```
Input: Texel Co-ordinates (tu,tv), Base - Starting address of Texture
Output: Texel address
$lbw \leftarrow log_2(bw)$
$lbh \leftarrow log_2(bh)$
$rs \leftarrow log_2(width \cdot bh)$
$bs \leftarrow log_2(bw \cdot bh)$
$bx \leftarrow tu >> lbw$
$by \leftarrow tv >> lbh$
$sx \leftarrow tu \&\& (bw - 1)$
$sy \leftarrow tv \&\& (bh - 1)$
block address $\leftarrow (by << rs) + (bx << bs)$
offset $\leftarrow (sy << lbw) + sx$
texel address $\leftarrow$ base + block address + offset
return texel address
```

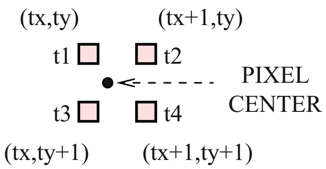


Fig. 7.21 Footprint of a Bilinear filter

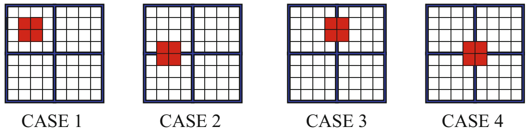


Fig. 7.22 Scenarios to which the bilinear filter footprint could be mapped: (i) CASE1: all four texels fall in same block, (ii) CASE2 and CASE3: the four texels fall in two of the neighboring block; and (iii) CASE4: the four texels fall in four different blocks

the four cases applies to a texture access can be obtained by comparing the block co-ordinates of the texels. If the accesses belong to *case 1*, where all the texels are mapped to same block, a lookup for texel $t1$ could be followed by fetching the texels $t2$, $t3$, and $t4$ from the same block. Thus, only one lookup suffices for fetching four texels. Similarly, for *cases 2* and *3*, two lookups are sufficient for four texel accesses. Only *case 4* requires four lookups.

Thus we see that though the mapping between pixel and texture space is determined only at run-time, the footprint of a bilinear filter always has a fixed pattern. Hence a customized memory architecture designed such that it allows both lookup operation as in a cache and also a direct register access to exploit spatial locality and predictability in the access stream results in a low power solution without compromising on performance. Texture Filter Memory (TFM), provides such access by introducing a few registers between the conventional texture cache and the texture fetch unit as shown in Fig. 7.23.

The texture blocks that are expected to be accessed in the near future are buffered in a set of registers, since register accesses need very little power compared to a

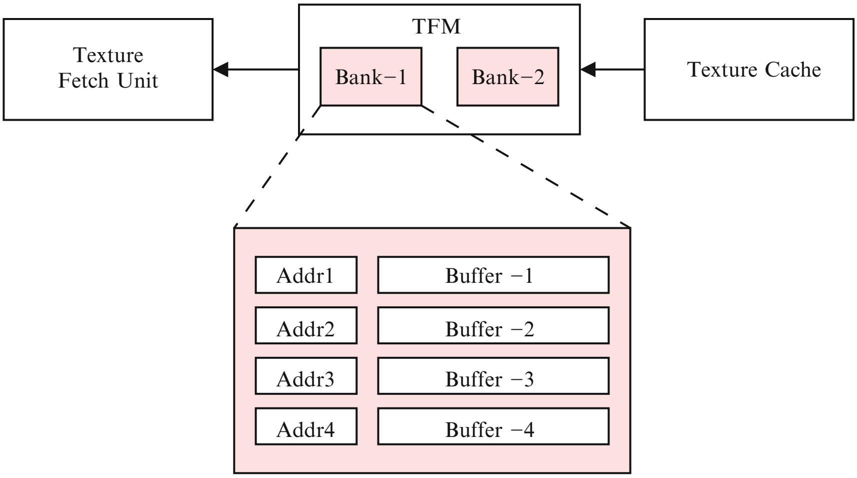


Fig. 7.23 Texture Filter Memory: each buffer stores a block of 4 × 4 texels. Each bank has 4 such buffers. Bilinear filtering needs one bank and trilinear filter needs two banks – one each for buffering values of two mipmap levels

standard data cache access. The number of blocks to be buffered depends on the type of filter being used. In a bilinear filtering operation, the texels could be in one, two, or four of the neighboring blocks as shown in Fig. 7.22. Also, since the next set of texels could fall in one of these four blocks with a high probability, upto four texture blocks are buffered for bilinear interpolation. In trilinear filtering we need to buffer eight blocks – four blocks from each of the two nearest mipmap levels.

The conventional kernel for bilinear filtering, as shown in Algorithm 3, needs to be modified to the one shown in Algorithm 4 so as to take advantage of the above buffering mechanism.

Algorithm 3 Kernel For Bilinear Filtering

```
Input: Texel Co-ordinates (tu,tv), Base - Starting address of Texture
Compute texel addresses corresponding to texel co-ordinates (tu,tv), (tu+1,tv), (tu,tv+1) and (tu+1,tv+1)
for $I = 1$ to 4 do
    $texelI \leftarrow$ CacheLookup(texeladdressI)
end for
$color \leftarrow$ WeightedAverage($texel1, texel2, texel3, texel4$)
return color
```

Though two additional comparison operations are used for classifying the accesses to different cases, at the same time, the number of block address computations and the texel address computations is reduced. Since all four texels fall in the same block with high probability, only one lookup is required for most texture accesses. Even though this single lookup can consume the same power as in an associative

Algorithm 4 Modified Bilinear Texture Filtering

```
Input: Texel Co-ordinates (tu,tv), Base - Starting address of Texture
Output: Color
$bx \leftarrow tu >> lbw$
$by \leftarrow tv >> lbh$
$bx1 \leftarrow (tu + 1) >> lbw$
$by1 \leftarrow (tv + 1) >> lbh$
$c0 \leftarrow (bx = bx1)?\ 0 : 1$
$c1 \leftarrow (by = by1)?\ 0 : 1$
Calculate offset1, offset2, offset3 and offset4
if $c0 = 0$ and $c1 = 0$ then
    compute block address1
    $texel1 \leftarrow$ LookupBuffer(block address1, offset1)
    Read texels 2,3 and 4 from the same block
else if $c0 = 0$ and $c1 = 1$ then
    compute block address1 and block address 3
    $texel1 \leftarrow$ LookupBuffer(block address1, offset1)
    Read texels 2 from the same block
    $texel3 \leftarrow$ LookupBuffer(block address3, offset3)
    Read texels 4 from the same block
else if $c0 = 1$ and $c1 = 0$ then
    compute block address1 and block address 2
    $texel1 \leftarrow$ LookupBuffer(block address1, offset1)
    Read texels 3 from the same block
    $texel2 \leftarrow$ LookupBuffer(block address2, offset2)
    Read texels 4 from the same block
else
    compute block addresses of all the four texels
    for $I = 1$ to 4 do
        $texelI \leftarrow$ LookupBuffer(blockaddressI, offsetI)
    end for
end if
$color \leftarrow$ WeightedAverage($texel1, texel2, texel3, texel4$)
return color
```

cache (though smaller in magnitude because the buffers have only 4 registers), the remaining three accesses do not require any lookup/comparison operation because the register containing the block is already known. On an average, the number of memory accesses and comparisons is drastically reduced.

7.3.2 Texture Compression

To reduce the power consumed in fetching textures from off-chip caches, the texture memory bandwidth is reduced by transferring compressed textures from off-chip texture memory to the texture cache. Since texture accesses have a very high impact on system performance, the main requirement of s texture compression system is that it should allow fast random access to texture data. Block compression schemes such as JPEG are not suitable for textures though they give high compression ratios. Since the accesses to texture memory are non-affine, it cannot be assured that the decompressed data is used up before the next block is fetched and decompressed. In cases where consecutive texture accesses alternate between a few texture blocks, the same block would have to fetched and decompressed multiple times resulting in increased block fetch and decompression overhead. Hence, we require compression schemes where the texels in a block can be decompressed independent of the other elements of the block. The *S3TC* compression technique is commonly used for this purpose [14].

In this technique, for a block of texels, two reference values and a few values generated by interpolation of the reference values are chosen such that each texel in the block can be approximated to one of the chosen values with least loss in accuracy. For example, if four values are to be used to represent the colors of a texture block, and c_0 and c_1 are the chosen reference values, two other colors (c_2 and c_3) are generated from interpolation of c_0 and c_1 as shown in Fig. 7.24. For each texel in the block, the closest of the colors among c_0 to c_3 is chosen. Thus, a 4×4 tile would require 2 reference values and 16 2-bit offsets to generate the interpolants from the reference values instead of 16 texel values. Based on this principle five modes of compression named DXT1…DXT5 have been proposed, with varying accuracy and compression ratios.

1. DXT1 gives the highest compression ratio among all the variants of DXT compression. Texels are usually represented as 32-bit values with the R,G,B,A components allotted 8-bits each. However, most of the time, textures do not need 32-bit accuracy. Hence, DXT1 uses 16-bit representation for RGB components (5:6:5) of the reference colors and allows a choice of 0 or 255 for transarency.

   The colors that could be generated from the two 16 bit reference values and 2 bits per texel which determine the weights for interpolation are shown below:

   If c_0 and c_1 are the reference values, the other two colors are calculated as
   If $c_0 > c_1$

$$c_2 = \frac{2c_0 + c_1}{3} \quad \text{and} \quad c_3 = \frac{c_0 + 2c_1}{3}$$

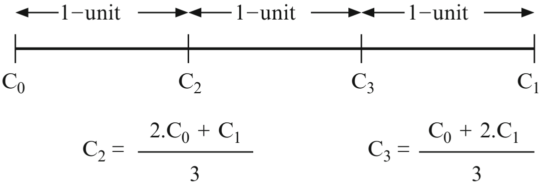


Fig. 7.24 c_2 and c_3 are generated on the fly by interpolation of c_0 and c_1

else

$$c_2 = \frac{c_0 + c_1}{2} \text{ and } c_3 = 0$$

For a 4 × 4 tile size, this scheme needs 64 bits per tile giving 8:1 compression.

2. The DXT2 and DXT3 compression schemes encode alpha values also in addition to color values and the compression scheme is similar to that described in DXT1. Thus for a 4 × 4 tile, they need 64 bits for color as in DXT1 and an additional 64 bits for alpha values, giving a 4:1 compression. In DXT2, color data is assumed to be pre-multiplied by alpha, which is not the case in DXT3.
3. In the DXT4 and DXT5 schemes, color components are compressed as in DXT2/3 and for alpha compression, two 8-bits reference values are used and 6 other alphas are interpolated from them, giving 8 alpha values to choose from. The alpha encoding is as shown below:

   If $\alpha_0 > \alpha_1$,

$$\alpha_2 = \frac{6\alpha_0 + \alpha_1}{7}, \quad \alpha_3 = \frac{5\alpha_0 + 2\alpha_2}{7}, \quad \alpha_4 = \frac{4\alpha_0 + 3\alpha_3}{7},$$

$$\alpha_5 = \frac{3\alpha_0 + 4\alpha_4}{7}, \quad \alpha_6 = \frac{2\alpha_0 + 5\alpha_5}{7}, \quad \alpha_7 = \frac{\alpha_0 + 6\alpha_6}{7}$$

   else

$$\alpha_2 = \frac{4\alpha_0 + \alpha_1}{5}, \quad \alpha_3 = \frac{3\alpha_0 + 2\alpha_2}{5}, \quad \alpha_4 = \frac{2\alpha_0 + 3\alpha_3}{5},$$

$$\alpha_5 = \frac{\alpha_0 + 4\alpha_4}{5}, \quad \alpha_6 = 0, \quad \alpha_7 = 255$$

   DXT4 is used in case color is pre-multiplied with alpha and DXT5 is used if it are not. DXT4/5 also give 4:1 compression, but produce superior results for alpha values.

7.3.3 Clock Gating

Clock gating is a powerful technique that could be used to conserve the power dissipated in a texture unit. The textures are generally stored in such a way that the odd mipmap and even mipmap levels map to different cache banks so that the texels could be fetched in parallel during tri-linear interpolation. Moreover, the addressing and filtering units are also present in pairs so that the texels could be filtered in parallel so as to facilitate faster texture sampling. However, when the texels are filtered in bilinear mode, half of these units and texture banks are idle. There could also be intervals during which the vertex or pixel shader threads would not be using texturing at all. Since the requirement of texture and the type of filter used is a part of the state information that the driver sends to the GPU, texture enable and filtering modes are set before the processing of the batch starts. Ideally, half of these units could be powered off when bilinear filtering is used and the entire texture module is switched off when texturing is not used. However, since the intervals between switching from one condition to another may not be large enough to merit powering-down of these circuits, clock gating is generally used to conserve the power associated with clocking these units.

7.4 Raster Operations

Raster operations are highly memory intensive since they need multiple reads and writes to the off-chip framebuffer. Since off-chip accesses are slow and power hungry, these framebuffer accesses affect the power consumption and also the performance of the system. Reducing the memory bandwidth between the GPU and framebuffer is therefore a very important power-performance optimization. Major techniques that are generally employed to reduce the required bandwidth between GPU and VRAM are:

- Re-use the data fetched from VRAM to the maximum extent before it is replaced by other data. Efforts made in this direction include extensive on-chip caching and blocked data accesses to maximize cache hits.
- Send compressed data to GPU from VRAM and decompress it on-chip so as to decrease the memory traffic. The decoder has to be simple enough so that the savings due to compressed data transfers dominates the decompression cost in terms of power and performance.

In this section we discuss the data compression strategies for memory bandwidth reduction of color and depth buffer.

7.4.1 Depth Buffer Compression

As described in Fig. 7.13, the fragments are generated in tiles to exploit spatial locality of accesses to color, depth, and texture data. Several tiles of depth information are cached in the on-chip *depth cache*; whenever there is a miss in this cache, a tile of data is fetched into it from the off-chip VRAM. To reduce the memory bandwidth due to these transfers, VRAM stores and transfers compressed tiles, which are decompressed on-the-fly before they are stored in the on-chip depth cache. Differential Differential Pulse Code Modulation (DDPCM) is one of the popular compression techniques used for depth buffer compression [12]. It is based on the principle that, since the depth values of fragments of a triangle are generated by interpolation of depth values of the vertices, if a tile is completely covered by a single triangle, the second order differentials of the depth values across the tile would all be zeroes.

The steps in the compression scheme are enumerated below:

1. Start with a tile of depth values. Assuming the tile is covered by a single triangle, the interpolated depth values in the tile would be as shown below:

Z	Z + ΔX	Z + 2 ΔX	Z + 3 ΔX
Z + ΔY	Z + ΔX + ΔY	Z + 2 ΔX + ΔY	Z + 3 ΔX + ΔY
Z + 2 ΔY	Z + ΔX + 2 ΔY	Z + 2 ΔX + 2ΔY	Z + 3 ΔX + 2 ΔY
Z + 3 ΔY	Z + ΔX + 3 ΔY	Z + 2 ΔX + 3 ΔY	Z + 3 ΔX + 3 ΔY

2. Compute the column wise first order differentials, and then repeat the same step to obtain column-wise second order differentials.
3. Follow it up with row-wise second order differential computation.

Z	Z + ΔX	Z + 2 ΔX	Z + 3 ΔX
ΔY	ΔY	ΔY	ΔY
ΔY	ΔY	ΔY	ΔY
ΔY	ΔY	ΔY	ΔY

Z	Z + ΔX	Z + 2 ΔX	Z + 3 ΔX
ΔY	ΔY	ΔY	ΔY
0	0	0	0
0	0	0	0

Z	ΔX	0	0
ΔY	0	0	0
0	0	0	0
0	0	0	0

We see that in the best case (i.e., when the triangle covers a tile), we need to store only one z value and two differentials. Thus, for an 8 × 8 block (which would originally need 64 × 32 bits), the compressed form would need 32 bits for the reference z value and 2 × 33 bits for the differentials. Since the depth values are generally interpolated at a higher precision than they are stored in, the second order differentials would be one among the values 0,-1, and 1. Hence two bits are required to encode the value of the second differential. Thus with an addition 61×2 bits for the second differentials, a total of $32 + 2 \times 33 + 61 \times 2 = 220$ bits would be required instead of 2048 bits. With the two bits used to represent the differentials, four values can be realized. Since only values 0,-1, and 1 are required to be realized, the fourth value can be used to indicate the case when the differentials take values other than 0, -1, and 1. In this case a fixed number of second order differentials are stored at higher precision and picked up in order each time a violation is indicated.

7.4.2 Color Buffer Compression

The transfers from color buffer to color cache are also done in tiles, so as to exploit spatial locality of accesses. Hence, block based compression schemes are used for these data transfers. Since color values are not always interpolated from vertex colors (they could be textured), the compression scheme used for depth buffer compression is not very efficient for color buffer compression. The difference between the color values of the neighboring pixels is small and this makes variable length encoding of the differences a suitable option for color buffer compression [20]. This compression technique is called exponent coding since the numbers are represented as $s(2^x - y)$, where s is the sign bit and $y \in [0, 2^{x-1} - 1]$. $x + 1$ is unary coded and concatenated with sign and y coded in normal binary coding to give the compressed value. For example, value 3 is represented as $(2^2 - 1)$. Here $x + 1 = 3$ which is 1110 in unary coding, $s = 0$ and $y = 1$, hence the code for 3 is 111001. Table 7.1 shows the coded values for numbers in the range $[-32, 32]$.

From the table we see that smaller numbers can be represented with relatively fewer number of bits than larger numbers. Since in most of the cases the differentials are observed to be small, significant compression ratios can be expected. Color values are used by both GPU and also the display controller. Compression helps reduce the bandwidth between the framebuffer and display controller also. Hence the display controller also needs a decompressor to decode the compressed color values read from the framebuffer.

Table 7.1 Exponential Encoding - Smaller numbers need lesser number of bits

Value range	Code
0_b	0
$10s_b$	± 1
$110s_b$	± 2
$1110sx_b$	$\pm[3, 4]$
$11110sxx_b$	$\pm[5, 8]$
$11110sxxx_b$	$\pm[9, 16]$
$11110sxxxx_b$	$\pm[17, 32]$
$11110sxxxxx_b$	8-bit absolute value

7.5 System Level Power Management

In addition to the architectural power optimization techniques discussed so far, system level power management techniques also prove to be effective in reducing the power consumption by minimizing the wastage of power in a graphics subsystem. Techniques such as system level power gating, V_{dd} and V_{th} scaling, and DVFS scaling are very efficient in saving power. These techniques, as applicable to GPUs, are discussed in detail in this section.

7.5.1 Power Modes

Graphics processors are used for accelerating various kinds of applications such as word processors, GUIs for various tools such as internet browsers, games, etc. Since the amount of graphics processing varies to a great extent from application to application, the GPU workload due to these applications also varies greatly. Moreover, there could be large intervals of time during which none of the applications requires graphics processing, leaving the GPU idle. Since it is not always required to operate the GPU at peak performance levels, a few power modes with varying performance levels are generally supported. For example, when the GPU is idle, it can be operated at minimum V_{dd} and V_{th} levels saving maximum dynamic and leakage power. However, when 3D games, which use heavy graphics processing are running on the system, the GPU can be operated at maximum performance mode. Performance monitors are used to gauge the utilization of the GPU (similar to monitoring CPU utilization), and the operating system switches the GPU to a power mode that delivers the required performance level with minimum power consumption.

7.5.2 Dynamic Voltage and Frequency Scaling

In case of power management by mode switching, since the switching overhead is high, there is a relatively large difference between the thresholds that cause a transition between power modes. The observation intervals are also large. However, applications such as games have been shown to exhibit significant variation in workload presented by different frames in the application. Fine tuning the computational capacity of the GPU in response to such workload variations has a huge power saving potential. DVFS techniques can be employed to achieve this with minimum performance impact. The properties of an application that makes it amenable to DVFS are:

1. varying workload, and
2. accurate predictability of workload.

Though games also show significant variation of workload from frame to frame, their interactive nature might give us the impression that it would be difficult to adapt the system to abrupt workload variations, making it unsuitable for DVFS. To understand why interactivity makes games distinct from any other real time application in the context of DVFS, let us see how DVFS is applied to a video decoder. The MPEG video decoding standard has been used extensively in literature to make a case for DVFS [6, 15].

A video stream is generally composed of a series of still images which, when displayed sequentially at a constant rate, creates an illusion of motion. These images are called frames, and are stored in compressed form so as to minimize the storage and transfer requirements. MPEG is a popular standard used for compression of video streams. MPEG compression divides the video stream into a sequence of *Group of Pictures* (GOPs), each of which comprises several frames. Each frame is further divided into vertical strips called slices and each slice is divided into several macro blocks which comprise a 16 × 16 pixel area of the image. Header information is associated with each structure in this hierarchical representation of the video stream; this information is used for workload prediction [23]. During the decoding process, all frames in a GOP are first buffered, the workload of the entire GOP is estimated, and the optimum value of voltage and frequency at which the GOP is to be decoded to meet the deadline is determined. Since the first frame is displayed only after the entire GOP is decoded, an output buffer is required to hold all the decoded frames of the GOP. Since the input frames keep streaming in at constant rate while decoding time could vary from frame to frame, an input buffer is used to store the incoming frames as shown in Fig. 7.25.

The example shown in Fig. 7.26 clearly demonstrates the advantage of buffer based DVFS over one without buffering [23]. If DVFS is applied over a set of buffered frames instead of varying the operating point online on per frame prediction basis, the slack over the set of frames can be accumulated and distributed among all the frames, leading to a lower power solution. An additional advantage of the buffer based mechanism is that it is possible to correct the losses due to mis-predictions to some extent in this method. Errors in predicting the workload would result in accumulation of frames in these buffers [4]. Hence the buffer occupancy is constantly monitored to correct the prediction inaccuracies and the operating voltage and frequency scaled accordingly as shown in Fig. 7.27.

In case of games, the application has to respond to the user's inputs, and ideally, the response needs to be instantaneous. For instance, when the user is playing a

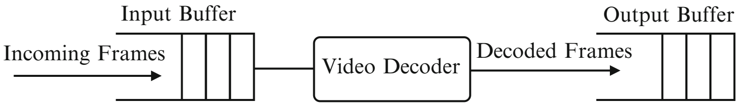


Fig. 7.25 I/O buffering for video decoder: since decoding time varies from frame to frame, the input buffer is needed to hold the frames streaming in. All the decoded frames are buffered into the output buffer until the GOP is completely rendered, and only then the decoded frames of the entire GOP are streamed out

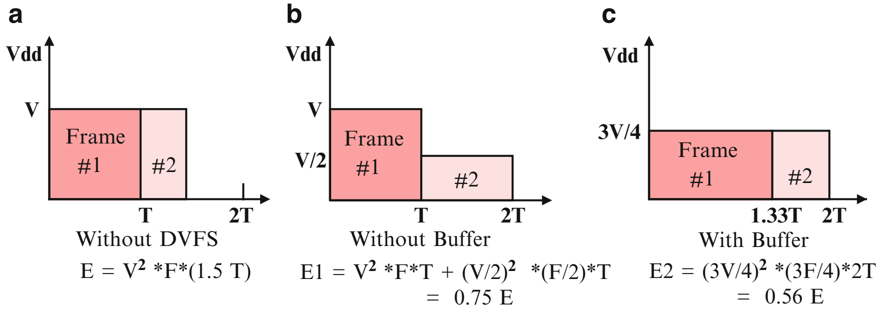


Fig. 7.26 Buffer based DVFS for video decoder. (**a**) A slack of T/2 is expected if the frames are processed at maximum frequency. (**b**) In this case, only frame 2 makes use of the slack. (**c**) The slack is distributed among both the buffered frames resulting in a lower power solution

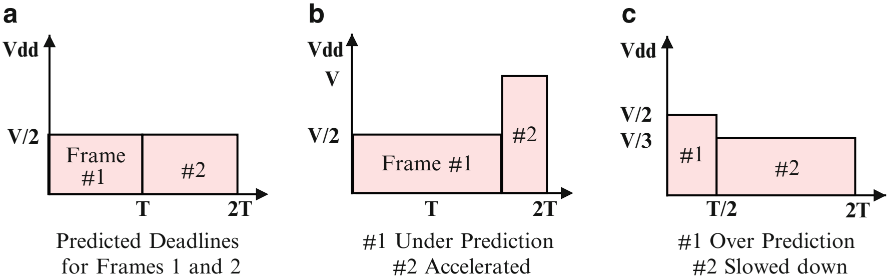


Fig. 7.27 Buffer occupancy based correction of workload prediction – (**a**) shows the predicted deadlines for frames 1 and 2 (**b**) workload for Frame 1 was underpredicted, hence the frame remains in output buffer for more than the expected time. This results in increased buffer occupancy. Frame 2 has to be processed faster to bring back the occupancy to the threshold value. (**c**) workload for Frame 1 was overpredicted, hence the frame leaves the output buffer at a rate faster than expected. Frame 2 can be slowed down to bring the occupancy rate to the threshold level

shooter game and has fired at a target, he would want this to reflect immediately on the screen rather than after some latency. Here we do not have the liberty to buffer a set of frames and then correct prediction inaccuracies of one frame by adjusting the operating voltage of the next frame. Interactivity also gives the impression that workload prediction based on previous frames behavior would not be viable for games. However, this is not true owing to the fact that consecutive frames of the games exhibit high levels of coherence in their pixel values. This is because, to maintain continuity of motion, positions of the objects in the frame can be displaced only by small amounts. Thus, the workloads do exhibit large but infrequent variations, making games excellent candidates for DVFS.

Since the *quality of service* in games is highly sensitive to the frame rates, it is important to predict the workload accurately in order to minimize the number of frames missing their deadlines. Some techniques use the workload history to predict the expected workload of the current frame, while others attempt to extract hints from the frame state information to guide the workload prediction. Various prediction techniques proposed in literature are discussed in more detail in the following sections.

7.5.2.1 History based Workload Estimation

The *history based workload estimation* technique predicts the workload of the current frame from the workload of the previously rendered frames [10]. The simplest and most straightforward way to do this is to approximate the workload of the current frame to that of the previous frame. However, doing so would result in frequent voltage-frequency changes, which are not desirable, since switching from one voltage-frequency level to another imposes an overhead of stabilization time. To minimize the number of transitions, the average workload of a window of previous frames is used to guess the workload of the current frame. A large window size is helpful in reducing the number of voltage changes, but at the same time, leads to a larger number of frames missing the deadlines as a result of the slower correction mechanism. This history based workload prediction can be extended to estimate the workload of all the voltage islands in the design and the operating point of each of the islands can be tuned to match the workload changes experienced by the island.

7.5.2.2 Control Theory based Workload Estimation

Control theory based DVFS takes into account the previous prediction error along with the previous predicted workload to predict the workload of the current frame [8]. Since it can adapt faster to the workload changes, it results in lesser number of frames missing their deadline. In a control based DVFS scheme, a simple Proportional Integral Derivative (PID) controller, as shown in Fig. 7.28, is used as

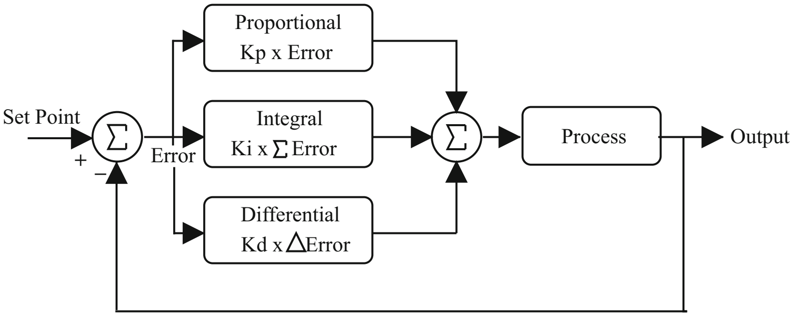


Fig. 7.28 PID controller: Proportional component aims to reduce prediction error; Integral component reduces the accumulated error in prediction; and the Differential component aims to reduce the rate of increase of the prediction error

a closed loop feedback mechanism to adjust the predicted workload of the current frame based on the prediction errors for some of the previously rendered frames. The workload of the current frame w_i is expressed as

$$w_i = w_{i-1} + delta(w) \tag{7.5}$$

where $delta(w)$ is the output from the PID controller. The proportional control regulates the speed at which the predicted workload responds to the prediction error of the previous frame. The Integral control determines how the workload prediction reacts to the prediction errors accumulated over a few of the recently processed frames. The differential control adjusts the workload based on the rate at which the prediction errors have changed over a few of the recent frames.

Thus the correction value generated by the PID controller can be expressed as

$$delta(w) = K_p \times Error + K_i \times \sum Error + K_d \times \Delta Error \tag{7.6}$$

The contribution of each of the Proportional, Integral, and Differential components of the controller can be tuned by varying the coefficients K_p, K_i, and K_d respectively. The flow of operations that take place in a PID based DVFS scheme can be summarized as shown in Fig. 7.29. Based on the difference between the actual workload and the predicted workload (Error) of the current frame, the PID controller estimates the workload for the next frame. The volatge and frequency of the system are scaled to match the computational capacity of the system with the predicted workload of the next frame. The frame is processed at this operating point and actual workload of the frame is observed to generate the Error value that drives the PID controller.

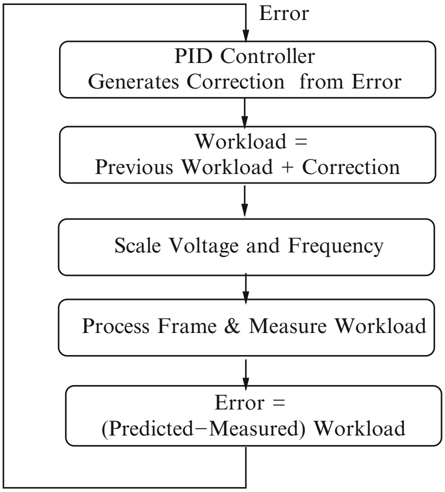

Error
PID Controller
Generates Correction from Error
Workload =
Previous Workload + Correction
Scale Voltage and Frequency
Process Frame & Measure Workload
Error =
(Predicted−Measured) Workload

Fig. 7.29 PID controller based DVFS for graphics processor

7.5.2.3 Frame Structure based Workload Estimation

In all the above discussed methods, the workload of a frame is estimated based on the history of previously processed frames. Hence the prediction would be good only when the scene remains almost the same across consecutive frame captures. The workload is bound to be mis-predicted when there is a significant change in the scene, which may result in frames missing their deadlines. To alleviate this problem, the *frame structure based estimation* technique bases its prediction on the structure of the frame and the properties of the objects present in the frame [9]. Since this information is obtained prior to processing of the frame, the workload prediction could be based on the properties of the current frame rather than basing it on the workload of previous frames. In this approach, a set of parameters impacting the workload are identified and an analytical model for the workload as a function of these parameters is constructed. During the execution of the application, each frame is parsed to obtain these parameters and the pre-computed workload model is used to predict the expected workload of the current frame. For example, the basic elements that make up a frame in the *Quake* game engine can be enumerated as follows.

- Brush models used to construct the world space. The complexity of a brush model is determined by the number of polygons present in the model. If the average workload for processing a polygon is w, the workload W presented by n brush models each consisting of p polygons is represented as:

$$W = n \times p \times w \tag{7.7}$$

- Alias models used to build the characters and objects such as monsters, soldiers, weapons, etc. Alias models consist of geometry and the skin texture of the entity being modeled. The skin could be rendered in one of two modes – opaque and alpha blend. Since the geometry consists essentially of triangles, its workload is characterized in terms of the number of triangles and average area of the triangle. Since alpha blending and opaque blending present different workloads, the workload is parametrized for both modes of rendering. If the workload of processing a single pixel with blending is w_t and without blending is w_o, the workload W due to alias models consisting of N_t triangles with blend and N_o opaque triangles, of average area A is given by:

$$W = N_t \times A \times w_t + N_o \times A \times w_o \tag{7.8}$$

- Textures applied to the surfaces of brush model to give a realistic appearance like that of wood, brick wall, etc. The workload W due to applying N_t textures, where w is the workload for applying a single texture on N polygons with average area A, is given by:

$$W = N_t \times N \times A \times w \tag{7.9}$$

- Light maps to create lighting effects in the scene. Since they are similar to texture maps, the workload due to light-maps is estimated similar to the estimation for texture maps.
- Particles to create bullets, debris, dust, etc. The workload W due to rendering the N particles, where the number of pixels in a particle i is given as P_i and workload for rendering one such pixel is w, is given by:

$$W = N \times P_i \times w \tag{7.10}$$

Finally the total workload of the frame is the sum total of the workloads computed above.

7.5.2.4 Signature based Workload Estimation

The *Signature based estimation* technique aims to estimate the workload using the properties of the frame in addition to the history of cycles expended in processing the previous frames [17]. Every frame is associated with a signature composed from its properties such as the number of triangles in the frame, average height and area of the triangles, the number of vertices in the frame, etc. A signature table records the actual observed workload of the frame against the signature of the frame. Prior to rendering a frame, its signature is computed and the predicted workload of the frame is picked from the signature table. On rendering the frame, if there is a discrepancy between the observed and predicted workloads, the signature table is updated with the observed workload value.

To compute the signature of the frame, we need the vertex count, triangle count, and also the area and height of the triangles. The pipeline has to be modified to facilitate the signature extraction since the triangle information can be obtained only after the triangle culling and clipping are performed. The modified pipeline is shown in Fig. 7.30. The geometry stage is divided into vertex transformation and lighting stages. Triangle clipping and culling stages are now performed prior to lighting and a signature buffer is inserted prior to the lighting stage to collect the fram statistics. Since we need the information of the entire frame to compute a meaningful signature, the buffer should be big enough to handle one frame delay. Signature based prediction works on the assumption that the computational intensity of the pre-signature stage is negligible and also can be performed on the CPU without hardware acceleration.

For every signature generated, the best matching signature from the table is to be looked up. A *distance* metric shown in equation 7.11 is used to locate the signature that is closest to the current signature. For a signature S consisting of parameters $s_1, s_2, ...s_d$ and a signature T comprising of $t_1, ...t_d$ in the signature table, the distance $D(S, T)$ is defined as

$$D(S,T) = \sum_{i=1}^{d} \frac{|s_i - t_i|}{s_i} \tag{7.11}$$

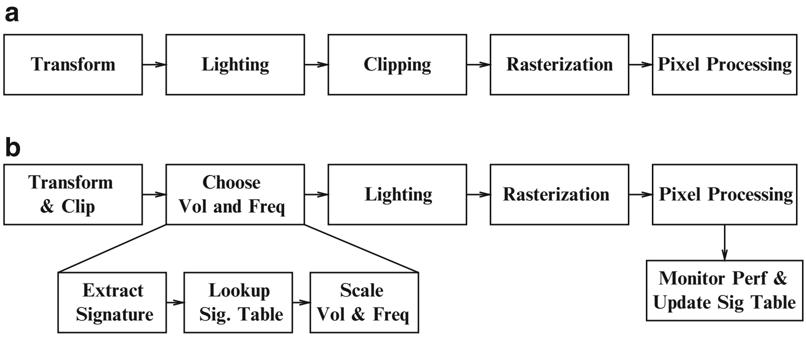


Fig. 7.30 (**a**) Conventional pipeline. (**b**) Signature based DVFS for graphics processor: the signature buffer collects the primitives of the frame after geometry processing. Once the primitives of the entire frame are collected in the signature buffer, the signature of the current frame is estimated and the DVFS operating point is selected based on the signature

The signature that is at a minimum distance from the current signature could be looked up either by linear search or any other sophisticated searching mechanism.

7.5.3 Multiple Power Domains

From the discussion in Section 7.1.3, it is clear that PUs, texture units, and ROPs are major components in the graphics processor that consume power. From the workload analysis of games it has been observed that some frames use a lot of texturing; others load the programmable units; still others require a large number of ROP operations. Hence these three modules could be designed to have different sets of power and clock signals. Thus, the voltage and frequency of each of these domains can be independently varied in accordance to their load, leading to power savings.

7.6 Summary

In this chapter we discussed a few important low power techniques used for graphics processors. There is ample scope to customize the architecture to the kind of processing in graphics applications in order to minimize power dissipation. In this chapter, we discussed the applicability of several low power techniques discussed earlier in the book in the context of power-efficient graphics processor design.

We studied memory customization for texture caches, compiler support for power optimized code generation for vertex shaders, clock and power gating for several blocks in the processor, along with system level power management techniques such as dynamic voltage scaling, switching to low power modes, and also multiple power domain design as mechanisms for improving power efficiency.

Graphics processor architectures are evolving extremely rapidly, and we expect several new power optimization possibilities will arise in the near future. The distinct architecture evolution paths of high-end GPUs discussed in this chapter and the relatively simpler mobile graphics processors, represent a good illustration of the performance-power trade-offs alluded in earlier chapters of the book.

References

1. Http://www.anandtech.com/
2. Barrio, V.M.D., González, C., Roca, J., Fernández, A., Espasa, R.: A single (unified) shader gpu microarchitecture for embedded systems. In: HiPEAC, pp. 286–301 (2005)
3. Chen, C.H., Lee, C.Y.: Two-level hierarchical Z-buffer for 3D graphics hardware. In: Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on, vol. 2, pp. II–253–II–256 vol.2 (2002)
4. Choi, K., Soma, R., Pedram, M.: Off-chip latency-driven dynamic voltage and frequency scaling for an mpeg decoding. In: DAC '04: Proceedings of the 41st annual Design Automation Conference, pp. 544–549. ACM, New York, NY, USA (2004). DOI http://doi.acm.org/10.1145/996566.996718
5. Chung, K., Yu, C.H., Kim, L.S.: Vertex cache of programmable geometry processor for mobile multimedia application. In: Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on, pp. 4 pp.– (2006). DOI 10.1109/ISCAS.2006.1692983
6. Flautner, K., Mudge, T.: Vertigo: automatic performance-setting for linux. SIGOPS Oper. Syst. Rev. **36**(SI), 105–116 (2002). DOI http://doi.acm.org/10.1145/844128.844139
7. Greene, N., Kass, M., Miller, G.: Hierarchical Z-buffer visibility. In: SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques, pp. 231–238. ACM, New York, NY, USA (1993). DOI http://doi.acm.org/10.1145/166117.166147
8. Gu, Y., Chakraborty, S.: Control theory-based dvs for interactive 3D games. In: DAC '08: Proceedings of the 45th annual Design Automation Conference, pp. 740–745. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1391469.1391659
9. Gu, Y., Chakraborty, S.: Power management of interactive 3D games using frame structures. In: VLSI Design, 2008. VLSID 2008. 21st International Conference on, pp. 679–684 (2008). DOI 10.1109/VLSI.2008.102
10. Gu, Y., Chakraborty, S., Ooi, W.T.: Games are up for dvfs. In: Design Automation Conference, 2006 43rd ACM/IEEE, pp. 598–603 (2006). DOI 10.1109/DAC.2006.229295
11. Hakura, Z.S., Gupta, A.: The design and analysis of a cache architecture for texture mapping. SIGARCH Comput. Archit. News **25**(2), 108–120 (1997). DOI http://doi.acm.org/10.1145/384286.264152
12. Hasselgren, J., Akenine-Möller, T.: Efficient depth buffer compression. In: GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, pp. 103–110. ACM, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1283900.1283917
13. Hoppe, H.: Optimization of mesh locality for transparent vertex caching. In: SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques, pp. 269–276. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1999). DOI http://doi.acm.org/10.1145/311535.311565
14. Iourcha, K.I., Nayak, K.S., Hong, Z.: System and method for fixed-rate block-based image compression with inferred pixel values. Patent 5956431 (1999). Http://www.freepatentsonline.com/5956431.html

15. Lu, Z., Hein, J., Humphrey, M., Stan, M., Lach, J., Skadron, K.: Control-theoretic dynamic frequency and voltage scaling for multimedia workloads. In: CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems, pp. 156–163. ACM, New York, NY, USA (2002). DOI http://doi.acm.org/10.1145/581630.581654
16. McCormack, J., McNamara, R.: Tiled polygon traversal using half-plane edge functions. In: HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, pp. 15–21. ACM, New York, NY, USA (2000). DOI http://doi.acm.org/10.1145/346876.346882
17. Mochocki, B.C., Lahiri, K., Cadambi, S., Hu, X.S.: Signature-based workload estimation for mobile 3D graphics. In: DAC '06: Proceedings of the 43rd annual Design Automation Conference, pp. 592–597. ACM, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1146909.1147062
18. Möller, T., Haines, E.: Real-time rendering. A. K. Peters, Ltd., Natick, MA, USA (1999)
19. Park, W.C., Lee, K.W., Kim, I.S., Han, T.D., Yang, S.B.: A mid-texturing pixel rasterization pipeline architecture for 3D rendering processors. In: Application-Specific Systems, Architectures and Processors, 2002. Proceedings. The IEEE International Conference on, pp. 173–182 (2002). DOI 10.1109/ASAP.2002.1030717
20. Rasmusson, J., Strom, J., Akenine-Moller, T.: Error-bounded lossy compression of floating-point color buffers using quadtree decomposition. Vis. Comput. **26**(1), 17–30 (2009). DOI http://dx.doi.org/10.1007/s00371-009-0372-y
21. Silpa, B., Kumar S.S, V., Panda, P.R.: Adaptive partitioning of vertex shader for low power high performance geometry engine. In: Advances in Visual Computing, Lecture Notes in Computer Science, vol. 5875/2009, pp. 111–124. Springer Berlin / Heidelberg (2009). DOI 10.1007/978-3-642-10331-5_11
22. Silpa, B.V.N., Patney, A., Krishna, T., Panda, P.R., Visweswaran, G.S.: Texture filter memory: a power-efficient and scalable texture memory architecture for mobile graphics processors. In: ICCAD '08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design, pp. 559–564. IEEE Press, Piscataway, NJ, USA (2008)
23. Tan, Y., Malani, P., Qiu, Q., Wu, Q.: Workload prediction and dynamic voltage scaling for mpeg decoding. In: ASP-DAC '06: Proceedings of the 2006 Asia and South Pacific Design Automation Conference, pp. 911–916. IEEE Press, Piscataway, NJ, USA (2006). DOI http://doi.acm.org/10.1145/1118299.1118505
24. Wang, P.H., Chen, Y.M., Yang, C.L., Cheng, Y.J.: A predictive shutdown technique for gpu shader processors. IEEE Computer Architecture Letters **8**, 9–12 (2009). DOI http://doi.ieeecomputersociety.org/10.1109/L-CA.2009.1

Index