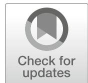
Coarse-Grained Reconfigurable Array Architectures



Bjorn De Sutter, Praveen Raghavan, and Andy Lambrechts

Abstract Coarse-Grained Reconfigurable Array (CGRA) architectures accelerate the same inner loops that benefit from the high instruction-level parallelism (ILP) support in very long instruction word (VLIW) architectures. Unlike VLIWs, CGRAs are designed to execute only the loops, which they can hence do more efficiently. This chapter discusses the basic principles of CGRAs and the wide range of design options available to a CGRA designer, covering a large number of existing CGRA designs. The impact of different options on flexibility, performance, and power-efficiency is discussed, as well as the need for compiler support. The ADRES CGRA design template is studied in more detail as a use case to illustrate the need for design space exploration, for compiler support, and for the manual fine-tuning of source code.

1 Application Domain of Coarse-Grained Reconfigurable Arrays

Many embedded applications require high throughput. At the same time, the power consumption of battery-operated devices needs to be minimized to increase their autonomy. In general, the performance obtained on a programmable processor for a certain application can be defined as the reciprocal of the application execution time. Considering that most programs consist of a series P of consecutive phases with different characteristics, performance can be defined in terms of the operating frequencies f_p, the instructions executed per cycle IPC_p and instruction count IC_p

B. De Sutter (✉)
Ghent University, Gent, Belgium
e-mail: bjorn.desutter@ugent.be

P. Raghavan · A. Lambrechts
imec, Heverlee, Belgium
e-mail: ragha@imec.be; lambreca@imec.be

© Springer International Publishing AG, part of Springer Nature 2019

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_12

of each phase, and in terms of the time overhead involved in switching between the phases $t_{p\rightarrow p+1}$ as follows:

$$\frac{1}{\text{performance}} = \text{execution time} = \sum_{p\in P} \frac{IC_p}{IPC_p \cdot f_p} + t_{p\rightarrow p+1}. \quad (1)$$

The operating frequencies f_p cannot be increased infinitely because of power-efficiency reasons. Alternatively, a designer can increase the performance by designing or selecting a system that can execute code at higher IPCs. In a power-efficient architecture, a high IPC is reached for the most important phases $l \in L \subset P$, with L typically consisting of the compute-intensive inner loops, while limiting their instruction count IC_l and reaching a sufficiently high, but still power-efficient frequency f_l. Furthermore, the time overhead $t_{p\rightarrow p+1}$ as well as the corresponding energy overhead of switching between the execution modes of consecutive phases should be minimized if such switching happens frequently. Note that such switching only happens on hardware that supports multiple execution modes in support of phases with different characteristics.

Course-Grained Reconfigurable Array (CGRA) accelerators aim for these goals for the inner loops found in many digital signal processing (DSP) domains. Such applications have traditionally employed Very Long Instruction Word (VLIW) architectures such as the TriMedia 3270 [112] and the TI C64 [106], Application-Specific Integrated Circuits (ASICs), and Application-Specific Instruction Processors (ASIPs). To a large degree, the reasons for running these applications on VLIW processors also apply for CGRAs. First of all, a large fraction of the computation time is spent in manifest nested loops that perform computations on arrays of data and that can, possibly through compiler transformations, provide a lot of Instruction-Level Parallelism (ILP). Secondly, most of those inner loops are relatively simple. When the loops include conditional statements, those can be implemented by means of predication [70] instead of control flow. Furthermore, none or very few loops contain multiple exits or continuation points in the form of, e.g., `break` or `continue` statements. Moreover, after inlining the loops are free of function calls. Finally, the loops are not regular or homogeneous enough to benefit from vector computing, like on the EVP [8] or on Ardbeg [113]. When there is enough regularity and Data-Level Parallelism (DLP) in the loops of an application, vector computing can typically exploit it more efficiently than what can be achieved by converting the DLP into ILP and exploiting that on a CGRA. So in short, CGRAs are ideally suited for applications of which time-consuming parts have manifest behavior, large amounts of ILP and limited amounts of DLP.

Over the last decade, applications from many domains have been accelerated on CGRAs. These include video processing [7, 17, 18, 67, 71, 100], image processing [40], audio processing [103], linear [76] and non-linear [69, 92] algebras, software-defined radios [11, 12, 25, 104, 110], augmented reality [85], biomedical applications [44], and Map-Reduce algorithms [62]. In support of these applications, CGRAs have also been commercialized. The Samsung Reconfigurable Processor,

the commercialized version of the ADRES CGRA that Samsung and imec initially developed as a proof-of-concept, has been used in ultra-high definition televisions and in smartphones, amongst others.

In the remainder of this chapter, Sect. 2 presents the fundamental properties of CGRAs. Section 3 gives an overview of the design options for CGRAs. This overview help designers in evaluating whether or not CGRAs are suited for their applications and their design requirements, and if so, which CGRA designs are most suited. After the overview, Sect. 4 presents a case study on the ADRES CGRA architecture. This study serves two purposes. First, it illustrates the extent to which source code needs to be tuned to map well onto CGRA architectures. As we will show, this is an important aspect of using CGRAs, even when good compiler support is available and when a very flexible CGRA is targeted, i.e., one that puts very few restrictions on the loop bodies that it can accelerate. Secondly, our use case illustrates how Design Space Exploration is necessary to instantiate optimized designs from parameterizable and customizable architecture templates such as the ADRES architecture template. Some conclusions are drawn in Sect. 5.

2 CGRA Basics

CGRAs focus on the efficient execution of the type of loops discussed in the previous section. By neglecting non-loop code or outer-loop code that is assumed to be executed on other cores, CGRAs can take the VLIW principles for exploiting ILP in loops a step further to consume less energy and deliver higher performance, without compromising on available compiler support. Figures 1 and 2 illustrate this.

Higher performance for high-ILP loops is obtained through two main features that separate CGRA architectures from VLIW architectures. First, CGRA architectures typically provide more Issue Slots (ISs) than typical VLIWs do. In the CGRA literature, some other commonly used terms to denote CGRA ISs are Arithmetic-Logic Units (ALUs), Functional Units (FUs), or Processing Elements (PEs). Conceptually, these terms all denote the same: logic on which an instruction can be executed, typically one per cycle. For example, a typical ADRES CGRA [11–13, 24, 71, 73–75] consists of 16 issue slots, whereas the TI C64 features 8 slots, and

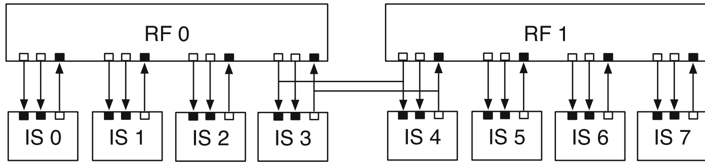


Fig. 1 An example clustered VLIW architecture with two RFs and eight ISs. Solid directed edges denote physical connections. Black and white small boxes denote input and output ports, respectively. There is a one-to-one mapping between input and output ports and physical connections

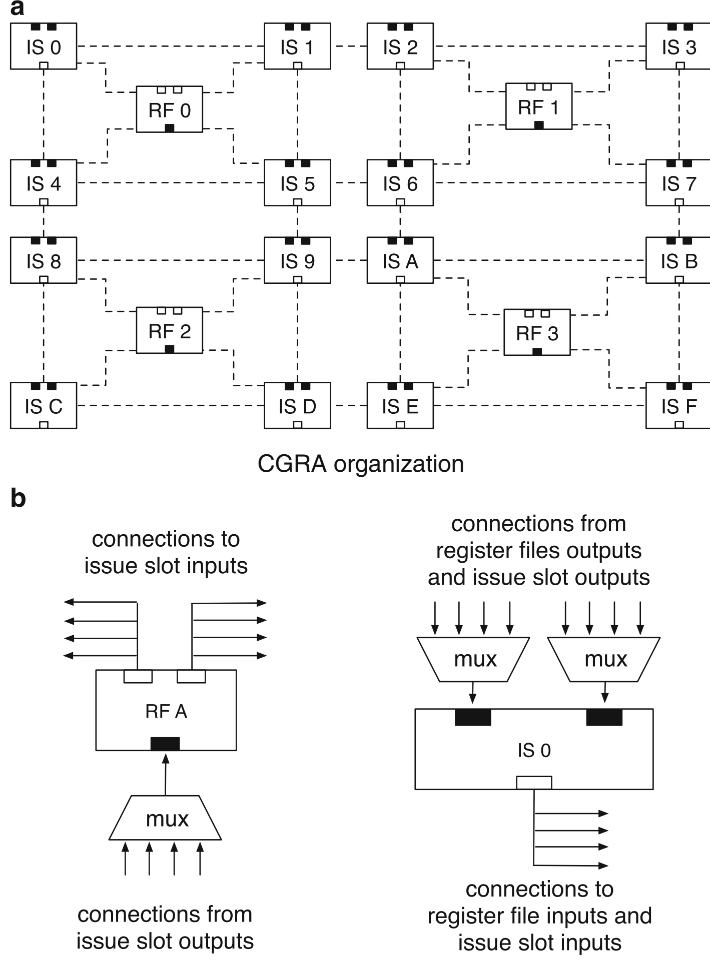


Fig. 2 Part (**a**) shows an example CGRA with 16 ISs and 4 RFs, in which dotted edges denote conceptual connections that are implemented by physical connections and muxes as in part (**b**)

the NXP TriMedia features only 5 slots. The higher number of ISs directly allows to reach higher IPCs, and hence higher performance, as indicated by Eq. (1). To support these higher IPCs, the bandwidth to memory is increased by having more load/store ISs than on a typical VLIW, and special memory hierarchies as found on ASIPs, ASICs, and other DSPs. These include FIFOs, stream buffers, scratch-pad memories, etc. Secondly, CGRA architectures typically provide a number of direct connections between the ISs that allow data to flow from one IS to another without

needing to pass data through a Register File (RF). As a result, less register copy operations need to be executed in the ISs, which reduces the IC factor in Eq. (1) and frees ISs for more useful computations.

Higher energy-efficiency is obtained through several features. Because of the direct connections between ISs, less data needs to be transferred into and out of RFs. This saves considerable energy. Also, because the ISs are arranged in a 2D matrix, small RFs with few ports can be distributed in between the ISs as depicted in Fig. 2. This contrasts with the many-ported RFs in (clustered) VLIW architectures, which basically feature a one-dimensional design as depicted in Fig. 1. The distributed CGRA RFs consume considerably less energy. Finally, by not supporting control flow, the instruction memory organization can be simplified. In statically reconfigurable CGRAs, this memory is nothing more than a set of configuration bits that remain fixed for the whole execution of a loop. Clearly this is very energy-efficient. Other CGRAs, called dynamically reconfigurable CGRAs, feature a form of distributed level-0 loop buffers [59] or other small controllers that fetch new configurations every cycle from simple configuration buffers. To support loops that include control flow and conditional operations, the compiler replaces that control flow by data flow by means of predication [70] or other mechanisms. In this way, CGRAs differ from VLIW processors that typically feature a power-hungry combination of an instruction cache, instruction decompression and decoding pipeline stages, and a non-trivial update mechanism of the program counter.

CGRA architectures have two main drawbacks. Firstly, because they only execute loops, they need to be coupled to other cores on which all other parts of the program are executed. This coupling can introduce run-time and design-time overhead. Secondly, as clearly visible in the example in Fig. 2, the interconnect structure of a CGRA is vastly more complex than that of a VLIW. On a VLIW, scheduling an instruction in some IS automatically implies the reservation of connections between the RF and the IS and of the corresponding ports. On CGRAs, this is not the case. Because there is no one-to-one mapping between connections and input/output ports of ISs and RFs, connections need to be reserved explicitly by the compiler or programmer together with ISs, and the data flow needs to be routed explicitly over the available connections. This can be done, for example, by programming switches and multiplexors (a.k.a. muxes) explicitly, like the ones depicted in Fig. 2b. Consequently more complex compiler technology than that of VLIW compilers [43] is needed to automate the mapping of code onto a CGRA. Moreover, writing assembly code for CGRAs ranges from being very difficult to virtually impossible, depending on the type of reconfigurability and on the form of processor control.

Having explained these fundamental concepts that differentiate CGRAs from VLIWs, we can now also differentiate them from Field-Programmable Gate Arrays (FPGAs), where the name CGRA actually comes from. Whereas FPGAs feature *bitwise* logic in the form of Look-Up Tables (LUTs) and switches, CGRAs feature more energy-efficient and area-conscious *word-wide* ISs, RFs, and interconnections. Hence the name coarse-grained array architecture. As there are much fewer ISs on a CGRA than there are LUTs on an FPGA, the number of bits required to configure

the CGRA ISs, muxes, and RF ports is typically orders of magnitude smaller than on FPGAs. If this number becomes small enough, dynamic reconfiguration can be possible every cycle. So in short, CGRAs can be seen as statically or dynamically reconfigurable coarse-grained FPGAs, or as 2D, highly-clustered loop-only VLIWs with direct interconnections between ISs that need to be programmed explicitly.

3 CGRA Design Space

The large design space of CGRA architectures features many design options. These include the way in which the CGRA is coupled to a main processor, the type of interconnections and computation resources used, the reconfigurability of the array, the way in which the execution of the array is controlled, support for different forms of parallelism, etc. This section discusses the most important design options and the influence of the different options on important aspects such as performance, power efficiency, compiler friendliness, and flexibility. In this context, higher flexibility equals placing fewer restrictions on loop bodies that can be mapped onto a CGRA.

Our overview of design options is not exhaustive. Its scope is limited to the most important features of CGRA architectures that feature a 2D array of ISs. However, the distinction between 1D VLIWs and 2D CGRAs is anything but well-defined. The reason is that this distinction is not simply a layout issue, but one that also concerns the topology of the interconnects. Interestingly, this topology is precisely one of the CGRA design options with a large design freedom.

3.1 Tight Versus Loose Coupling

Some CGRA designs are coupled loosely to main processors. For example, Fig. 3 depicts how the MorphoSys CGRA [60] is connected as an external accelerator to a TinyRISC Central Processing Unit (CPU) [1]. The CPU is responsible for executing non-loop code, for initiating DMA data transfers to and from the CGRA and the buffers, and for initiating the operation of the CGRA itself by means of special instructions added to the TinyRISC ISA.

This type of design offers the advantage that the CGRA and the main CPU can be designed independently, and that both can execute code concurrently, thus delivering higher parallelism and higher performance. For example, using the double frame buffers [60] depicted in Fig. 3, the MorphoSys CGRA can be operating on data in one buffer while the main CPU initiates the necessary DMA transfers to the other buffer for the next loop or for the next set of loop iterations. One drawback is that any data that needs to be transferred from non-loop code to loop code needs to be transferred by means of DMA transfers. This can result in a large overhead, e.g., when frequent switching between non-loop code and loops with few iterations occurs and when the loops consume scalar values computed by non-loop code.

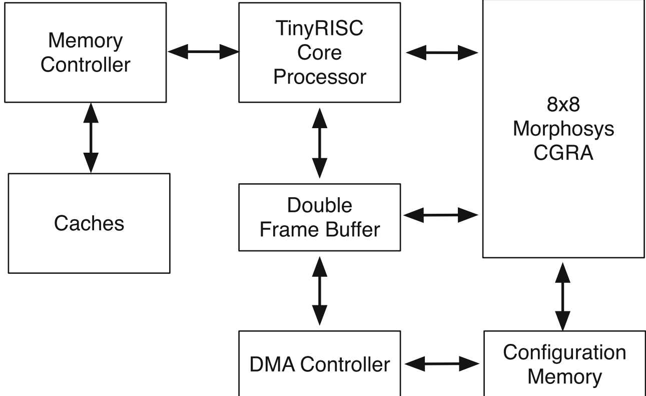


Fig. 3 A TinyRISC main processor loosely coupled to a MorphoSys CGRA array. Note that the main data memory (cache) is not shared and that no IS hardware or registers is are shared between the main processor and the CGRA. Thus, both can run concurrent threads

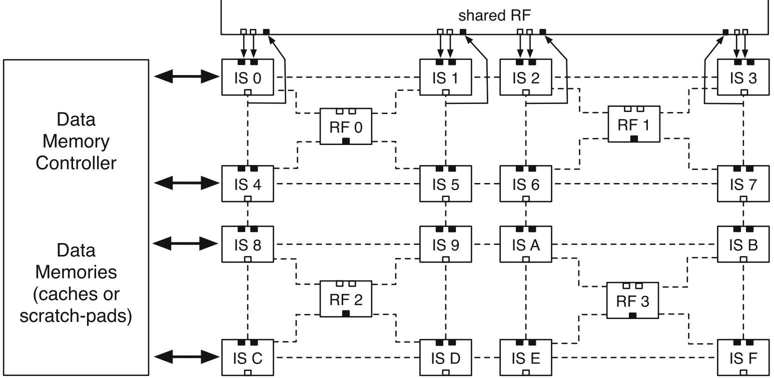


Fig. 4 A simplified picture of an ADRES architecture. In the main processor mode, the top row of ISs operates like a VLIW on the data in the shared RF and in the data memories, fetching instructions from an instruction cache. When the CGRA mode is initiated with a special instruction in the main VLIW ISA, the whole array starts operating on data in the distributed RFs, in the shared RF and in the data memories. The memory port in IS 0 is also shared between the two operating modes. Because of the resource sharing, only one mode can be active at any point in time

By contrast, an ADRES CGRA is coupled tightly to its main CPU. A simplified ADRES is depicted in Fig. 4. Its main CPU is a VLIW consisting of the shared RF and the top row of CGRA ISs. In the main CPU mode, this VLIW executes instructions that are fetched from a VLIW instruction cache and that operate on data in the shared RF. The idle parts of the CGRA are then disabled by clock-gating to save energy. By executing a `start_CGRA` instruction, the processor switches to CGRA mode in which the whole array, including the shared RF and the top row of ISs, executes a loop for which it gets its configuration bits from a configuration memory. This memory is omitted from the figure for the sake of simplicity.

The drawback of this tight coupling is that because the CGRA and the main processor mode share resources, they cannot execute code concurrently. However, this tight coupling also has advantages. Scalar values that have been computed in non-loop code, can be passed from the main CPU to the CGRA without any overhead because those values are already present in the shared RFs or in the shared memory banks. Furthermore, using shared memories and an execution model of exclusive execution in either main CPU or CGRA mode significantly eases the automated co-generation of main CPU code and of CGRA code in a compiler, and it avoids the run-time overhead of transferring data between memories. Finally, on the ADRES CGRA, switching between the two modes takes only two cycles. Thus, the run-time overhead is minimal. That overhead can still be considerable, however, in the case of nested loops, as inner loops are then entered and exited many times. Moreover, upon entry and exit of a software pipelined loop, resources are wasted as the software pipeline fills and drains in the so-called prologue and epilogue of the loop. This will be discussed in more detail in Sect. 4.1.1. Two design extensions have been proposed to reduce this overhead. First, instruction set extensions have been proposed to reduce the overhead that flattening of imperfectly nested loops introduces [57]. By flattening loop nests, less mode switches are necessary. Secondly, the Remus CGRA design for streaming data applications features an array in which the data flows in one direction, i.e., from one row to another, top to bottom [66, 117, 118]. The rows of the CGRA then operate as if they are a statically scheduled pipeline. During the epilogue of one loop, the rows gradually become unused by that loop, and hence they become available for the next loop to be executed. The next loop's prologue can hence start executing as soon as the current loop's epilogue has started. In many applications, this can save considerable execution time.

Silicon Hive CGRAs [14, 15] do not feature a clear separation between the CGRA accelerator and the main processor. Instead there is just a single processor that can be programmed at different levels of ILP, i.e., at different instruction word widths. This allows for a very simple programming model, with all the programming and performance advantages of the tight coupling of ADRES. Compared to ADRES, however, the lack of two distinctive modes makes it more difficult to implement coarse-grained clock-gating or power-gating, i.e., gating of whole sets of ISs combined instead of separate gating of individual ISs.

Somewhere in between loose and tight coupling is the PACT XPP design [79], in which the array consist of simpler ISs that can operate like a true CGRA, as well as of more complex ISs that are in fact full-featured small RISC processors that can run independent threads in parallel with the CGRA.

As a general rule, looser coupling potentially enables more Thread-Level Parallelism (TLP) and it allows for a larger design freedom. Tighter coupling can minimize the per-thread run-time overhead as well as the compile-time overhead. This is in fact no different from other multi-core or accelerator-based platforms.

3.2 CGRA Control

Many different mechanisms exist to control how code gets executed on CGRAs, i.e., to control which operation is issued on which IS at which time and how data values are transferred from producing operations to consuming ones. Two important aspects of CGRAs that drive different methods for control are reconfigurability and scheduling. Both can be static, dynamic, or a hybrid combination thereof.

3.2.1 Reconfigurability

Some CGRAs, like ADRES, Silicon Hive, and MorphoSys are fully dynamically reconfigurable: Exactly one full reconfiguration takes place for every execution cycle. Of course no reconfiguration takes places in cycles in which the whole array is stalled. Such stalls can happen, e.g., because memory accesses take longer than expected in the schedule as a result of a cache miss or a memory bank access conflict. This cycle-by-cycle reconfiguration is similar to the fetching of one VLIW instruction per cycle, but on these CGRAs the fetching is simpler as it only iterates through a loop body existing of straight-line CGRA configurations without control flow. Other CGRAs like the KressArray [37–39] are fully statically reconfigurable, meaning that the CGRA is configured before a loop is entered, and no reconfiguration takes place during the loop at all. Still other architectures feature a hybrid reconfigurability. The RaPiD [22, 27] architecture features partial dynamic reconfigurability, in which part of the bits are statically reconfigurable and another part is dynamically reconfigurable and controlled by a small sequencer. Yet another example is the PACT architecture, in which the CGRA itself can initiate events that invoke (partial) reconfiguration. This reconfiguration consumes a significant amount of time, however, so it is advised to avoid it if possible, and to use the CGRA as a statically reconfigurable CGRA.

In statically reconfigured CGRAs, each resource performs a single task for the whole duration of the loop. In that case, the mapping of software onto hardware becomes purely spatial, as illustrated in Fig. 5a. In other words, the mapping problem becomes one of placement and routing, in which instructions and data dependencies between instructions have to mapped on a 2D array of resources. For these CGRAs, compiler techniques similar to hardware synthesis techniques can be used, as those used in FPGA placement and routing [9].

By contrast, dynamic reconfigurability enables the programmer to use hardware resources for multiple different tasks during the execution of a loop or even during the execution of a single loop iteration. In that case, the software mapping problem becomes a spatial and temporal mapping problem, in which the operations and data transfers not only need to be placed and routed on and over the hardware resources, but in which they also need to be scheduled. A contrived example of a temporal mapping is depicted in Fig. 5b. Most compiler techniques [24, 26, 29, 73, 75, 78, 81, 82, 107] for these architectures also originate from the FPGA placement and routing

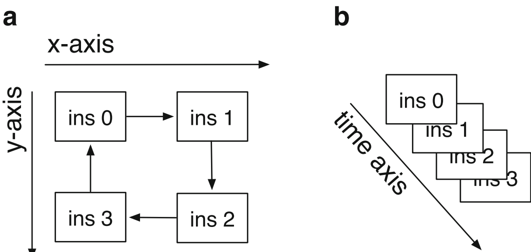


Fig. 5 Part (**a**) shows a spatial mapping of a sequence of four instructions on a statically reconfigurable 2×2 CGRA. Edges denote dependencies, with the edge from instruction 3 to instruction 0 denoting that instruction 0 from iteration i depends on instruction 3 from iteration $i - 1$. So only one out of four ISs is utilized per cycle. Part (**b**) shows a temporal mapping of the same code on a dynamically reconfigurable CGRA with only one IS. The utilization is higher here, at 100%

world. For CGRAs, the array of resources is not treated as a 2D spatial array, but as a 3D spatial-temporal array, in which the third dimension models time in the form of execution cycles. Scheduling in this dimension is often based on techniques that combine VLIW scheduling techniques such as modulo scheduling [43, 54, 93], with FPGA synthesis-based techniques [9]. Still other compiler techniques exist that are based on constraint solving [101], or on integer linear programming [2, 56, 127].

The most important advantage of static reconfigurability is the lack of reconfiguration overhead, in particular in terms of power consumption. For that reason, large arrays can be used that are still power-efficient. The disadvantage is that even in the large arrays the amount of resources constrains which loops can be mapped.

Dynamically reconfigurable CGRAs can overcome this problem by spreading the computations of a loop iteration over multiple configurations. Thus a small dynamically reconfigurable array can execute larger loops. The loop size is then not limited by the array size, but by the array size times the depth of the reconfiguration memories. For reasons of power efficiency, this depth is also limited, typically to tens or hundreds of configurations, which suffices for most if not all inner loops.

A potential disadvantage of dynamically reconfigurable CGRAs is the power consumption of the configuration memories, even for small arrays, and of the configuration fetching mechanism. The disadvantage can be tackled in different ways. ADRES and MorphoSys tackle it by not allowing control flow in the loop bodies, thus enabling the use of very simple, power-efficient configuration fetching techniques similar to level-0 loop buffering [59]. Whenever control flow is found in loop bodies, such as for conditional statements, this control flow then first needs to be converted into data flow, for example by means of predication and hyperblock formation [70]. While these techniques can introduce some initial overhead in the code, this overhead typically will be more than compensated by the fact that a more efficient CGRA design can be used.

The MorphoSys design takes this reduction of the reconfiguration fetching logic even further by limiting the supported code to Single Instruction Multiple Data

(SIMD) code. In the two supported SIMD modes, all ISs in a row or all ISs in a column perform identical operations. As such only one IS configuration needs to be fetched per row or column. As already mentioned, the RaPiD architecture limits the number of configuration bits to be fetched by making only a small part of the configuration dynamically reconfigurable. Kim et al. provide yet another solution in which the configuration bits of one column in one cycle are reused for the next column in the next cycle [52]. Furthermore, they also propose to reduce the power consumption in the configuration memories by compressing the configurations [53].

Still, dynamically reconfigurable designs exist that put no restrictions on the code to be executed, and that even allow control flow in the inner loops. The Silicon Hive design is one such design.

A general rule is that a limited reconfigurability puts more constraints on the types and sizes of loops that can be mapped. Which design provides the highest performance or the highest energy efficiency depends, amongst others, on the variation in loop complexity and loop size present in the applications to be mapped onto the CGRA. With large statically reconfigurable CGRAs, it is only possible to achieve high utilization for all loops in an application if all those loops have similar complexity and size, or if they can be made so with loop transformations, and if the iterations are not dependent on each other through long-latency dependency cycles (as was the case in Fig. 5). Dynamically reconfigurable CGRAs, by contrast, can also achieve high average utilization over loops of varying sizes and complexities, and with inter-iteration dependencies. That way dynamically reconfigurable CGRAs can achieve higher energy efficiency in the data path, at the expense of higher energy consumption in the control path. Which design option is the best depends also on the process technology used, and in particular on the ability to perform clock or power gating and on the ratio between active and passive power (a.k.a. leakage).

In that regard, it is interesting to note the recent research direction of so-called dual-V_{dd} and multi-V_{dd} CGRA designs [33, 115, 120] that goes beyond the binary approach of gating. In these designs, the supply voltage fed to different parts of a CGRA, which can be individual ISs or clusters thereof, can vary independently. This resembles dynamic voltage scaling as found on many modern multi-core CPUs, but in the case of CGRAs the supply voltages fed to a part of the CGRA is determined by the length of the critical path in the circuit that is triggered by the specific operations executing on that part of the array.

3.2.2 Scheduling and Issuing

Both with dynamic and with static reconfigurability, the execution of operations and of data transfers needs to be controlled. This can be done statically in a compiler, similar to the way in which operations from static code schedules are scheduled and issued on VLIW processors [28, 43], or dynamically, similar to the way in which out-of-order processors issue instructions when their operands become available [99]. Many possible combinations of static and dynamic reconfiguration and of static and dynamic scheduling exist.

A first class consists of dynamically scheduled, dynamically reconfigurable CGRAs like the TRIPS architecture [32, 95]. For this architecture, the compiler determines on which IS each operation is to be executed and over which connections data is to be transferred from one IS to another. So the compiler performs placement and routing. All scheduling (including the reconfiguration) is dynamic, however, as in regular out-of-order superscalar processors [99]. TRIPS mainly targets general-purpose applications, in which unpredictable control flow makes the generation of high-quality static schedules difficult if not impossible. Such applications most often provide relatively limited ILP, for which large arrays of computational resources are not efficient. So instead a small, dynamically reconfigurable array is used, for which the run-time cost of dynamic reconfiguration and scheduling is acceptable.

A second class of dynamically reconfigurable architectures avoids the overhead of dynamic scheduling by supporting VLIW-like static scheduling [28]. Instead of doing the scheduling in hardware where the scheduling logic then burns power, the scheduling for ADRES, MorphoSys and Silicon Hive architectures is done by a compiler. Compilers can do this efficiently for loops with regular, predictable behavior and high ILP, as found in many DSP applications. As is the case for VLIW architectures, software pipelining [43, 54, 93] is very important to expose the ILP in CGRA software kernels, so most compiler techniques [19, 24, 26, 29, 34, 35, 73, 75, 78, 81, 82, 107, 128] for statically scheduled CGRAs implement some form of software pipelining.

A third class of CGRAs are the statically reconfigurable, dynamically scheduled architectures, such as KressArray or PACT (neglecting the time-consuming partial reconfigurability of the PACT). The compiler performs placement and routing, and the code execution progress is guided by tokens or event signals that are passed along with data. Thus the control is dynamic, and it is distributed over the token or event path, similar to the way in which transport-triggered architectures [21] operate. These statically reconfigurable CGRAs do not require software pipelining techniques because there is no temporal mapping. Instead the spatial mapping and the control implemented in the tokens or event signals implement a hardware pipeline.

Hybrid designs exist as well. Park et al. use tokens not to trigger the execution of instructions, but to enable an opcode compression scheme without increasing decoder complexity with the end goal of reducing the power consumption [84]. In their statically scheduled CGRA, data-producing ISs send a token to the consuming ISs over a token datapath that complements the existing datapath. Based on the tokens that arrive, the consuming IS then already knows which type of operation it will need to execute, so less opcode bits need to be retrieved and decoded to program the IS. This way, they were able to obtain a 56% power reduction in the control path.

Another form of hybrid designs are the so-called triggered execution and dual-issue designs [36, 125, 126]. These are scheduled statically, but feature extensions to increase the resource utilization of loops bodies containing if-then-else structures. With standard predication techniques, the instructions of both the then and else branches occupy ISs. So in every iteration, the ISs used for the non-occupied branch are wasted. With the trigger-based and dual-issue extensions, two operations (one

from the then branch and one from the else branch) can be loaded together to configure the same IS, and additional predicate logic decides dynamically which of the operations is actually executed.

We can conclude by noting that, as in other architecture paradigms such as VLIW processing or superscalar out-of-order execution, dynamically scheduled CGRAs can deliver higher performance than statically scheduled ones for control-intensive code with unpredictable behavior. On dynamically scheduled CGRAs the code path that gets executed in an iteration determines the execution time of that iteration, whereas on statically scheduled CGRAs, the combination of all possible execution paths (including the slowest path which might be executed infrequently) determines the execution time. Thus, dynamically scheduled CGRAs can provide higher performance for some applications. However, the power-efficiency will then typically also be poor because more power will be consumed in the control path. Again, the application domain determines which design option is the most appropriate.

3.2.3 Thread-Level and Data-Level Parallelism

Another important aspect of control is the possibility to support different forms of parallelism. Obviously, loosely-coupled CGRAs can operate in parallel with the main CPU, but one can also try to use the CGRA resources to implement SIMD or to run multiple threads concurrently within the CGRA.

When dynamic scheduling is implemented via distributed event-based control, as in KressArray or PACT, implementing TLP is relatively simple and cheap. For small enough loops of which the combined resource use fits on the CGRA, it suffices to map independent thread controllers on different parts of the distributed control.

For architectures with centralized control, the only option to run threads in parallel is to provide additional controllers or to extend the central controller, for example to support parallel execution modes. While such extensions will increase the power consumption of the controller, the newly supported modes might suit certain code fragments better, thus saving in data path energy and configuration fetch energy.

The TRIPS controller supports four operation modes [95]. In the first mode, all ISs cooperate for executing one thread. In the second mode, the four rows execute four independent threads. In the third mode, fine-grained multi-threading [99] is supported by time-multiplexing all ISs over multiple threads. Finally, in the fourth mode each row executes the same operation on each of its ISs, thus implementing SIMD in a similar, fetch-power-efficient manner as is done in the two modes of the MorphoSys design. Thus, for each loop or combination of loops in an application, the TRIPS compiler can exploit the most suited form of parallelism.

The Raw architecture [105] is a hybrid between a many-core architecture and a CGRA architecture in the sense that it does not feature a 2D array of ISs, but rather a 2D array of tiles that each consist of a simple RISC processor. The tiles are connected to each other via a mesh interconnect, and transporting data over

this interconnect to neighboring tiles does not consume more time than retrieving data from the RF in the tile. Moreover, the control of the tiles is such that they can operate independently or synchronized in a lock-step mode. Thus, multiple tiles can cooperate to form a dynamically reconfigurable CGRA. A programmer can hence partition the 2D array of tiles into several, potentially differently sized, CGRAs that each run an independent thread. This provides very high flexibility to balance the available ILP inside threads with the TLP of the combined threads.

The Polymorphic Pipeline Array (PPA) [83] and similar designs [110] integrate multiple tightly-coupled ADRES-like CGRA cores into a larger array. Independent threads with limited amounts of ILP can run on the individual cores, but the resources of those individual cores can also be configured to form larger cores, on which threads with more ILP can then be executed. The utilization of the combined resources can be optimized dynamically by configuring the cores according to the available TLP and ILP at any point during the execution of a program.

Other architectures do not support (hardware) multi-threading within one CGRA core at all, like the Silicon Hive. The first solution to run multiple threads with these designs is to incorporate multiple CGRA accelerator cores in a System-on-Chip (SoC) [116]. The advantage is then that each accelerator can be customized for a certain class of loop kernels.

Alternatively, TLP can be converted into ILP and DLP by combining, at compile-time, kernels of multiple threads and by scheduling them together as one kernel, and by selecting the appropriate combination of scheduled kernels at run time [96].

3.3 Interconnects and Register Files

3.3.1 Connections

A wide range of connections can connect the ISs of a CGRA with each other, with the RFs, with other memories and with IO ports. Buses, point-to-point connections, and crossbars are all used in various combinations and in different topologies.

For example, some designs like MorphoSys and the most common ADRES and Silicon Hive designs feature a densely connected mesh-network of point-to-point interconnects in combination with sparser buses that connect ISs further apart. Thus the number of long power-hungry connections is limited. Multiple studies of point-to-point mesh-like interconnects as in Fig. 6 have been published in the past [13, 51, 55, 72]. Other designs like RaPiD feature a dense network of segmented buses. Typically the use of crossbars is limited to very small instances because large ones are too power-hungry. Fortunately, large crossbars are most often not needed, because many application kernels can be implemented as systolic algorithms, which map well onto mesh-like interconnects as found in systolic arrays [90].

Unlike crossbars and even busses, mesh-like networks of point-to-point connections scale better to large arrays without introducing too much delay or power consumption. For statically reconfigurable CGRAs, this is beneficial. Buses and

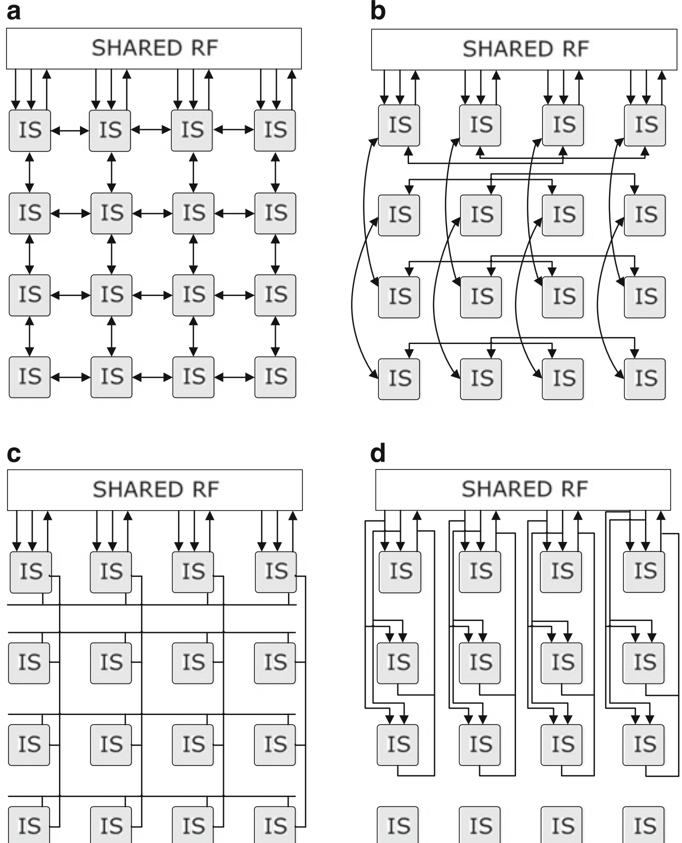


Fig. 6 Basic interconnects that can be combined. All bidirectional edges between two ISs denote that all outputs of one IS are connected to the inputs of the other IS and vice versa. Buses that connect all connected IS outputs to all connected IS inputs are shown as edges without arrows. (**a**) Nearest neighbor (nn), (**b**) next hop (nh), (**c**) buses (b), (**d**) extra (ex)

other long interconnects connect whole rows or columns to complement short-distance mesh-like interconnects. The negative effects that such long interconnects can have on power consumption or on obtainable clock frequency can be avoided by segmentation or by pipelining. In the latter case, pipelining latches are added along the connections or in between muxes and ISs. Our experience, as presented in Sect. 4.2.2 is that this pipelining will not necessarily lead to lower IPCs in CGRAs. This is different from out-of-order or VLIW architectures, where deeper pipelining increases the branch misprediction latency [99]. Instead at least some CGRA compilers succeed in exploiting the pipelining latches as temporary storage, rather than being hampered by them. This is the case in compiler techniques like [24, 73, 107] that are based on FPGA synthesis methods in which RFs and pipelining latches are treated as interconnection resources that span multiple cycles

instead of as explicit storage resources. This treatment naturally fits the 3D array modeling of resources along two spatial dimensions and one temporal dimension. Consequently, those compiler techniques can use pipelining latches for temporary storage as easily as they can exploit distributed RFs. This ability to use latches for temporary storage has been extended even beyond pipeline latches, for example to introduce retiming chains and shift registers in CGRA architectures [108].

As was already discussed in Sect. 3.1, the Remus architecture has an interconnect that lets data flow from top to bottom through an array. This fits streaming data applications, it simplifies the interconnect to potentially yield lower power consumption and higher clock speeds, and it enables to overlapping execution of one loop's epilogue with the next loop's prologue [125, 126].

3.3.2 Register Files

CGRA compilers place operations on ISs, thus also scheduling them, and route the data flow over the connections between the ISs. Those connections may be direct connections, or latched connections, or even connections that go through RFs. Therefore most CGRA compilers treat RFs not as temporary storage, but as interconnects that can span multiple cycles. Thus the RFs can be treated uniformly with the connections during routing. A direct consequence of this compiler approach is that the design space freedom of interconnects extends to the placement of RFs in between ISs. During the Design Space Exploration (DSE) for a specific CGRA instance in a CGRA design template such as the ADRES or Silicon Hive templates, both the real connections and the RFs have to be explored, and that has to be done together. Just like the number of real interconnect wires and their topology, the size of RFs, their location and their number of ports then contribute to the interconnectivity of the ISs. We refer to [13, 72] for DSEs that study both RFs and interconnects.

Besides their size and ports, another important aspect is that RFs can be rotating [94]. The power and delay overhead of rotation is very small in distributed RFs, simply because these RFs are small themselves. Still they can provide an important functionality. Consider a dynamically reconfigurable CGRA on which a loop is executed that iterates over x configurations, i.e., each iteration takes x cycles. That means that for a write port of an RF, every x cycles the same address bits get fetched from the configuration memory to configure the address set at that port. In other words, every x cycles a new value is being written into the register specified by that same address. This implies that values can stay in the same register for at most x cycles; then they are overwritten by a new value from the next iteration. In many loops, however, some values have a life time that spans more than x cycles, because it spans multiple loop iterations. To avoid having to insert additional data transfers in the loop schedules, rotating registers can be used. At the end of every iteration of the loop, all values in rotating registers rotate into another register to make sure that old values are copied to where they are not overwritten by newer values.

3.3.3 Predicates, Events and Tokens

To complete this overview on CGRA interconnects, we want to point out that it can be very useful to have interconnects of different widths. The data path width can be as small as 8 bits or as wide as 64 or 128 bits. The latter widths are typically used to pass SIMD data. However, as not all data is SIMD data, not all paths need to have the full width. Moreover, most CGRA designs and the code mapped onto them feature signals that are only one or a few bits wide, such as predicates or events or tokens. Using the full-width datapath for these narrow signals wastes resources. Hence it is often useful to add a second, narrow datapath for control signals like tokens or events and for predicates. How dense that narrow datapath has to be, depends on the type of loops one wants to run on the CGRA. For example, multimedia coding and decoding typically includes more conditional code than SDR baseband processing. Hence the design of, e.g., different ADRES architectures for multimedia and for SDR resulted in different predicate data paths being used, as illustrated in Sect. 4.2.1.

At this point, it should be noted that the use of predicates is fundamentally not that different from the use of events or tokens. In KressArray or PACT, events and tokens are used, amongst others, to determine at run time which data is selected to be used later in the loop. For example, for a C expression like `x + (a>b) ? y + z : y - z` one IS will first compute the addition `y+z`, one IS will compute the subtraction `y-z`, and one IS will compute the greater-than condition `a>b`. The result of the latter computation generates an event that will be fed to a multiplexor to select which of the two other computer values `y+z` and `y-z` is transferred to yet another IS on which the addition to `x` will be performed. Unlike the muxes in Fig. 2b that are controlled by bits fetched from the configuration memory, those event-controlled multiplexors are controlled by the data path.

In the ADRES architecture, the predicates guard the operations in ISs, and they serve as enable signals for RF write ports. Furthermore, they control special `select` operations that pass one of two input operands to the output port of an IS. Fundamentally, an event-controlled multiplexor performs exactly the same function as the `select` operation. So the difference between events or tokens and predicates is really only that the former term and implementation are used in dynamically scheduled designs, while the latter term is used in static schedules. As was already pointed out in Sect. 3.2.2, dual-issue and triggered instruction CGRAs combine the two forms to obtain higher resource utilization in the case of if-then-else structures.

3.4 Computational Resources

Issue slots are the computational resources of CGRAs. Over the last decade, numerous designs of such issue slots have been proposed, under different names, that include PEs, FUs, ALUs, and flexible computation components. Figure 7 depicts some of them. For all of the possible designs, it is important to know the

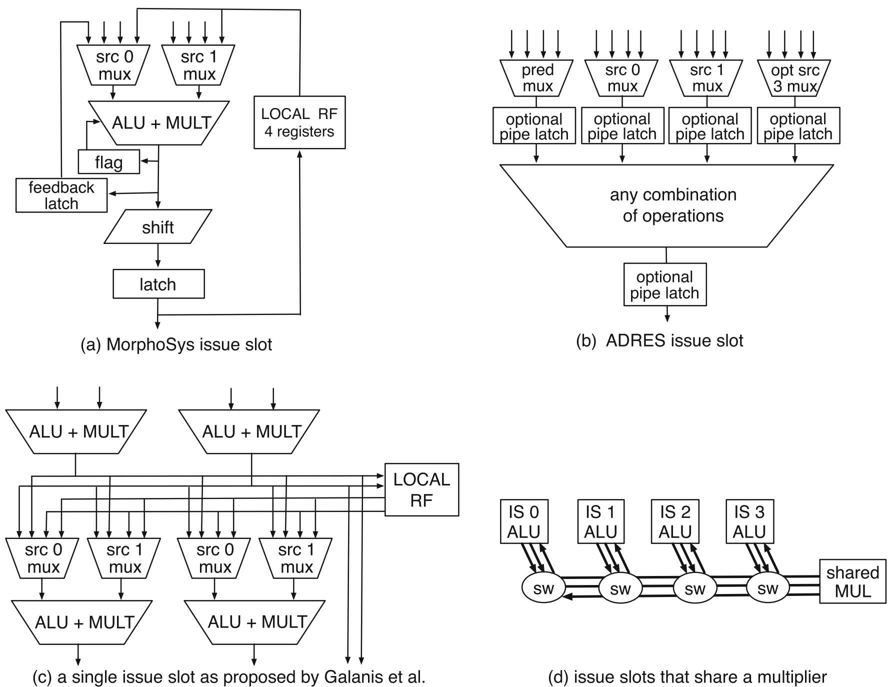


Fig. 7 Four different structures of ISs proposed in the literature. Part (**a**) displays a fixed MorphoSys IS, including its local RF. Part (**b**) displays the fully customizable ADRES IS, that can connect to shared or non-shared local RFs. Part (**c**) depicts the IS structure proposed by Galanis et al. [31], and (**d**) depicts a row of four RSPA ISs that share a multiplier [48]

context in which these ISs have to operate, such as the interconnects connecting them, the control type of the CGRA, etc.

Figure 7a depicts the IS of a MorphoSys CGRA. All 64 ISs in this homogeneous CGRA are identical and include their own local RF. This is no surprise, as the two MorphoSys SIMD modes (see Sect. 3.2.1) require that all ISs of a row or of a column execute the same instruction, which clearly implies homogeneous ISs.

In contrast, almost all features of an ADRES IS, as depicted in Fig. 7b, can be chosen at design time, and can be different for each IS in a CGRA that then becomes heterogeneous: the number of ports, whether or not there are latches between the multiplexors and the combinatorial logic that implements the operations, the set of operations supported by each IS, how the local registers file are connected to ISs and possibly shared between ISs, etc. As long as the design instantiates the ADRES template, the ADRES tool flow will be able to synthesize the architecture and to generate code for it. A similar design philosophy is followed by the Silicon Hive tools. Of course this requires more generic compiler techniques than those that generate code for the predetermined homogeneous ISs of, e.g., the MorphoSys CGRA. As we will discuss later in Sect. 3.6.2, this typically implies much longer

compilation times. Moreover, we need to note that while extensive specialization will typically benefit performance, it can also have negative effects, in particular on energy consumption [109].

Figure 7c depicts the IS proposed by Galanis et al. [31]. Again, all ISs are identical. In contrast to the MorphoSys design, however, these ISs consist of several ALUs and multipliers with direct connections between them and their local RFs. These direct connections within each IS can take care of a lot of data transfers, thus freeing time on the shared bus-based interconnect that connects all ISs. Thus, the local interconnect within each IS compensates for the lack of a scaling global interconnect. One advantage of this clustering approach is that the compiler can be tuned specifically for this combination of local and global connections and for the fact that it does not need to support heterogeneous ISs. Whether or not this type of design is more power-efficient than that of CGRAs with more design freedom and potentially more heterogeneity is unclear at this point in time. At least, we know of no studies from which, e.g., utilization numbers can be derived that allow us to compare the two approaches.

Some architectures combine the flexibility of heterogeneous ADRES ISs with clustering. For example, the CGRA Express [86] and the expression-grained reconfigurable array (EGRA) [3] feature heterogeneous clusters of relatively simple, fast ALUs. Within the clusters, those ALUs are chained by means of a limited number of latchless connections. Through careful design, the delay of those chains is comparable to the delay of other, more complex ISs on the CGRA that bound the clock frequency. So the chaining does not effect the clock frequency. It does allow, however, to execute multiple dependent operations within one clock cycle. It can therefore improve performance significantly. As the chains and clusters are composed of existing components such as ISs, buses, multiplexers and connections, these clustered designs do not really extend the design space of non-clustered CGRAs like ADRES. Still it can be useful to treat clusters as a separate design level in between the IS component level and the whole array architecture level, for example because it allows code generation algorithms in compilers to be tuned for there existence [86].

A specific type of clustering was proposed to handle floating-point arithmetic. While most CGRAs are limited to integer and fixed-point arithmetic, Lee at al. proposed to cluster two ISs to handle floating-point data [56]. In their design, both ISs in the cluster can operate independently on integer or fixed-point data, but they can also cooperate by means of a special direct interconnect between them. When they cooperate, one IS in the cluster consumes and handles the mantissas, while the other IS consumes and produces the exponents. As a single ISs can thus be used for both floating-point and integer computations, Lee et al. are able to achieve high utilization for integer applications, floating-point applications, as well as mixed applications.

Yet another type of clustering was proposed by Suh et al. [103]. They build a larger CGRA out of identical clusters, not to enable faster compilation or to obtain better performance, but to limit the time needed to perform design space explorations in order to reduce the time to market.

With respect to utilization, it is clear that the designs of Fig. 7a, b will only be utilized well if a lot of multiplications need to be performed. Otherwise, the area-consuming multipliers remain unused. To work around this problem, the sharing of large resources such as multipliers between ISs has been proposed in the RSPA CGRA design [48]. Figure 7d depicts one row of ISs that do not contain multipliers internally, but that are connected to a shared multiplier through switches and a shared bus. The advantage of this design, compared to an ADRES design in which each row features three pure ALU ISs and one ALU+MULT IS, is that this design allows the compiler to schedule multiplications in all ISs (albeit only one per cycle), whereas this scheduling freedom would be limited to one IS slot in the ADRES design. To allow this schedule freedom, however, a significant amount of resources in the form of switches and a special-purpose bus need to be added to the row. While we lack experimental data to back up this claim, we firmly believe that a similar increase in schedule freedom can be obtained in the aforementioned 3+1 ADRES design by simply extending an existing ADRES interconnect with a similar amount of additional resources. In the ADRES design, that extension would then also be beneficial to operations other than multiplications.

The optimal number of ISs for a CGRA depends on the application domain, on the reconfigurability, as well as on the IS functionality and on the DLP available in the form of subword parallelism. As illustrated in Sect. 4.2.2, a typical ADRES would consist of 4×4 ISs [12, 71]. TRIPS also features 4×4 ISs. MorphoSys provides 8×8 ISs, but that is because the DLP is implemented as SIMD over multiple ISs, rather than as subword parallelism within ISs.

In our experience, scaling dynamically reconfigurable CGRA architectures such as ADRES to very large arrays (8×8 or larger) is rarely useful, even with scalable interconnects like mesh or mesh-plus interconnects. Even in loops with high ILP, utilization drops significantly on such large arrays [77]. It is not clear what causes this lower utilization, and there might be several reasons. These include a lack of memory bandwidth, the possibility that the compiler techniques [24, 73] simply do not scale to such large arrays, or the fact that the relative connectivity in such large arrays is lower. Simply stated, when a mesh interconnects all ISs to their neighbors, each IS not on the side of the array is connected to 4 other ISs out of 16 in a 4×4 array, i.e., to 25% of all ISs, while it is connected to 4 out of 64 ISs on an 8×8 array, i.e., to 6.25% of all ISs.

Of course, large arrays can still be useful, e.g., if they can be partitioned in smaller arrays to run multiple threads in parallel, as discussed in Sect. 3.2.3. Also in CGRAs with limited connectivity, such as the Remus design introduced in Sect. 3.1, larger cores have proven useful.

3.5 Memory Hierarchies

CGRAs have a large number of ISs that need to be fed with data from the memory. Therefore the data memory sub-system is a crucial part of the CGRA design. Many

reconfigurable architectures feature multiple independent memory banks or blocks to achieve high data bandwidth.

The RAW architecture features an independent memory block in each tile for which Barua developed a method called modulo unrolling to disambiguate and assign data to different banks [5]. However, this technique can only handle array references through affine index expression on loop induction variables.

MorphoSys has a 256-bit wide frame buffer between the main memory and a reconfigurable array to feed data to the ISs operating in SIMD mode [60]. The efficient use of such a wide memory depends by and large on manual data placement and operation scheduling. Similar techniques for wide loads and stores have also been proposed in regular VLIW architectures for reducing power [91]. Exploiting that hardware requires manual data layout optimizations as well.

Both Silicon Hive and PACT feature distributed memory blocks without a crossbar. A Silicon Hive programmer has to specify the allocation of data to the memory for the compiler to bind the appropriate load/store operations to the corresponding memories. Silicon Hive also supports the possibility of interfacing the memory or system bus using FIFO interfaces. This is efficient for streaming processing but is difficult to interface when the data needs to be buffered on in case of data reuse.

The ADRES architecture template provides a parameterizable Data Memory Queue (DMQ) interface to each of the different single-ported, interleaved level-1 scratch-pad memory banks [23]. The DMQ interface is responsible for resolving bank access conflicts, i.e., when multiple load/store ISs would want to access the same bank at the same time. Connecting all load/store ISs to all banks through a conflict resolution mechanism allows maximal freedom for data access patterns and also maximal freedom on the data layout in memory. The potential disadvantage of such conflict resolution is that it increases the latency of load operations. In software pipelined code, however, increasing the individual latency of instructions most often does not have a negative effect on the schedule quality, because the compiler can hide those latencies in the software pipeline. In the main processor VLIW mode of an ADRES, the same memories are accessed in code that is not software-pipelined. So in that mode, the conflict resolution is disabled to obtain shorter access latencies.

Alternatively, a data cache can be added to the memory hierarchy to complement the scratch-pad memories. By letting the compiler partition the data over the scratch-pad memories and the data cache in an appropriate manner, high throughput can be obtained in the CGRA mode, as well as low latency in the VLIW mode [41, 45]. On a SoC with multiple CGRAs, the caches can be shared, and cache partitioning can be used to ensure that each CGRA obtains high throughput [116].

Furthermore, small local memories can be added exclusively to the CGRA to store data temporarily to lower the pressure on register files [124]. This way, memory hierarchies in CGRAs show many similarities to those found in modern Graphics Processing Units (GPUs). This should not be surprising. Samsung has already hinted that they plan to start using their Samsung Reconfigurable Processor designs in their future generations of GPUs [61]. Next to those GPU-like features, other features are adopted from high-level CPU designs. For example, data prefetch-

ing mechanisms have been proposed based on the history of the loop nests executed in CGRA mode [119].

3.6 *Compiler Support*

Two lines of research have to be discussed with respect to compiler support. The oldest one concerns the scheduler in the compiler back-end. This scheduler is responsible for determining where and when the operations of a loop body will be executed, and how data will flow through the interconnect from one IS to another. In some cases, it is also responsible for register allocation.

The other, more recent line of research concerns intermediate code generation and the optimization of intermediate code. This is the phase of the compiler that transforms the intermediate code to obtain loop bodies that are better suited to be mapped onto the targeted CGRAs, i.e., for which the scheduler can generate more efficient code.

3.6.1 Intermediate Code Generation and Optimization

In order to enable the back-end's scheduler to generate efficient code, i.e., code that utilizes the available resources of a CGRA well, some conditions need to be met: The loop bodies need to contain sufficient operations to utilize all the resources, the data dependencies between the operations need to enable high ILP, the memory access patterns should not create bottlenecks, as much as possible time has to be spent in inner loops, etc.

To obtain such loop bodies, compiler middle-ends apply loop transformations, such as flattening and unrolling [4]. For well-formed loop nests, such as affine ones, algebraic models are available, so-called polyhedral models [42], to reason about the degrees of freedom that a compiler has for reordering the operations in loop nests and to decide on the best transformation strategy for each loop. Such models have been used in parallelizing compilers of all kinds since about two decades [6]. The boundary conditions are somewhat different for CGRA compilers, however: entering and exiting CGRA mode results in considerably more overhead than doing so on general-purpose CPUs or VLIW processors and the number of available resources to be exploited through ILP is much higher. In Sect. 4.1, we will discuss these in more detail, when we discuss loop transformations for the ADRES CGRA template as a use case.

In CGRA programming environments that lack automated CGRA-specific loop optimization strategies, manual fine tuning of loops by rewriting their source code is therefore necessary to obtain acceptable code quality. Over the last couple of years, however, a range of automated loop optimization strategies has been developed that specifically target CGRAs and that can hence result in much more productive programming. Of those strategies, many rely on polyhedral models

and integer-linear programming for optimally merging affine or other perfect loop nests [64, 65, 68, 122, 123] and imperfectly nested loop nests [63, 121]. Others focus on determining the best loop unrolling parameters [98]. Whereas the aforementioned techniques focus on optimizing performance, some polyhedral techniques also consider battery conservation for mobile applications [88, 89].

3.6.2 CGRA Code Mapping and Scheduling Techniques

Apart from the specific algorithms used to schedule code, the major distinctions between CGRA schedulers relate to whether or not they support static scheduling, whether or not they support dynamic reconfiguration, whether or not they rely on special programming languages, and whether or not they are limited to specific hardware properties, or are instead flexible enough to support, e.g., very heterogeneous instances within an architecture template. Because most compiler research has been done to generate static schedules for CGRAs, we focus on those in this section. As already indicated in Sects. 3.2.1 and 3.2.2, many algorithms are based on FPGA placement and routing techniques [9] in combination with VLIW code generation techniques like modulo scheduling [54, 93] and hyperblock formation [70].

Whether or not compiler techniques rely on specific hardware properties is not always obvious in the literature, as not enough details are available in the descriptions of the techniques, and few techniques have been tried on a wide range of CGRA architectures. For that reason, it is very difficult to compare the efficiency (compilation time), the effectiveness (quality of generated code) and the flexibility (e.g., support for heterogeneity) of the different techniques.

The most widely applicable static scheduling techniques use different forms of Modulo Resource Routing Graphs (MRRGs). RRGs are time-space graphs, in which all resources (space dimension) are modeled with vertices. There is one such vertex per resource per cycle (time dimension) in the schedule being generated. Directed edges model the connections over which data values can flow from resource to resource. The schedule, placement, and routing problem then becomes a problem of mapping the Data Dependence Graph (DDG) of some loop body on the RRG. Scheduling refers to finding the right cycle to perform an operation (i.e., a DDG node) in the schedule, placement refers to finding the right IS (i.e., MRRG vertex) in that cycle, and routing refers to finding connections to transfer data from producing operations to consuming operations, i.e., to find a route in the MRRG for a DDG edge. In the case of a modulo scheduler, the modulo constraint is enforced by modeling all resource usage in the modulo time domain. This is done by modeling the appropriate modulo reservation tables [93] on top of the RRG, hence the name MRRG.

The granularity of its vertices depends on the precise compiler algorithm. One modulo graph embedding algorithm [81] for ADRES-like CGRAs models whole ISs or whole RFs with single vertices, whereas the simulated-annealing technique

in the DRESC [24, 73, 75] compiler that also targets ADRES instances models individual ports to ISs and RFs as separate vertices. Typically, fewer nodes that model larger components lead to faster compilation because the graph mapping problem operates on a smaller graph, but also to lower code quality because some combinations of resource usage cannot be modeled precisely. Moreover, models with fewer nodes also lack the flexibility to model a wide variation in resources, and hence can typically not model heterogeneous designs.

Several types of modulo schedulers for CGRAs exist. In the aforementioned DRESC, simulated annealing is used to explore different placement and routing options until a valid placement and routing of all operations and data dependencies is found. The cost function used during the simulated annealing is based on the total routing cost, i.e., the combined resource consumption of all placed operations and of all routed data dependencies. In this technique, a huge number of possible routes is evaluated, as a result of which the technique is very slow: Scheduling individual loops can take tens of minutes.

Later modulo scheduling techniques [29, 47, 78, 81, 82, 107] for ADRES-like CGRAs operate much more like (modulo) list schedulers [28]. These list-based CGRA schedulers still target MRRG representations of the hardware, and thus offer a large amount of flexibility in the architectures they support. Like DRESC, they rely heavily on routing costs. However, whereas DRESC first places DDG nodes in an MRRG and then tries to find good routes for the DDG edges connecting the nodes, these list schedulers work the opposite way. When one node of a DDG edge has already been placed (e.g., its sink node), a good place for the other node (the source node) is found by finding the cheapest possible path for the DDG edge in the MRRG, starting from the place of the already placed node. So in this case, the scheduler first identifies a good route for a DDG edge, and that route determines where its DDG node is placed. These schedulers are therefore called edge-centric schedulers. To find the best (i.e., cheapest) routes, they use a myriad of cost functions. These functions assign costs to nodes in the MRRG such that nodes that should not yet be occupied at a certain point during the iterative scheduling, e.g., because they model scarce resources that need to remain available for placing other DDG nodes later during the scheduling, are considered expensive and are hence avoided during the searches for cheapest routes. After every placement of a DDG node, the cost functions are updated in function of the next node to be placed, the nodes already placed and their places, the available resources, and the amounts and types of resources that will still be needed in the future. For some types of cost functions, these updates are simple, but for others they are very complex and computing them is time-consuming. The second [78] and third [107] generation edge-centric schedulers outperform the others in terms of generated code quality and compilation time because they offer a better balance between (1) cost function complexity; (2) priority functions, i.e., the order in which nodes are chosen to be placed onto the MRRG; and (3) their backtracing heuristics, i.e., the cases in which they unplace DDG nodes to try alternative places after a placement was found to block the generation of a valid, high quality schedule. The currently best scheduler even offers several modes of operation, in which fast, inaccurate cost functions are

tried first, and only if those fail, the slower, more accurate ones are used [107]. This delivers better code quality than DRESC can deliver, in particular for more heterogeneous CGRA designs, while requiring about 2 orders of magnitude less compilation time.

Several other graph-based modulo schedulers have been proposed that build on heavily simplified resource graphs to model the CGRA [19, 34, 35, 128]. Using different customized algorithms to find limited forms of sub-graph isomorphisms between a loop's DDG and the architecture resource graph, these schedulers can generate schedules very quickly. However, the limitation to certain forms of sub-graph isomorphisms can result in significantly lower code quality. Moreover, the simplified resource graphs cannot express many kinds of heterogeneity and features, such as varying places of latches in the CGRA. So these publications only consider rather homogeneous designs, in which only the supported instruction classes vary per IS. Some algorithms even seem to rely on the (in our view unrealistic) assumption that all operations have the same latency [34, 35].

Kim et al. presented a scheduler in which the generic NP-hard problem of modulo scheduling becomes tractable by imposing the constraint of following pre-calculated patternized rules [46]. As expected, the compilation times are improved by several orders of magnitude, at the cost of code quality (−30% compared to the already badly performing, first-generation edge-centric technique of [82]). Through its use of patternized rules, this scheduler is by construction limited to mostly homogeneous CGRAs. Lee et al. present an integer linear programming approach and a quantum-inspired evolutionary algorithm, both applied after an initial list scheduling [56]. Their mapping algorithms adopt high-level synthesis techniques combined with loop unrolling and software pipelining. They also target homogeneous targets.

MRRG-based compiler techniques are easily retargetable to a wide range of architectures, such as those of the ADRES template, and they can support many programming languages. Different architectures can simply be modeled with different MRRGs. It has even been demonstrated that by using the appropriate modulo constraints during the mapping of a DDG on a MRRG, compilers can generate a single code version that can be executed on CGRAs of different sizes [87]. This is particularly interesting for the PPA architecture that can switch dynamically between different array sizes [83] to support either a single big loop executing in a single threads or multiple smaller loops executing in parallel threads as discussed in Sect. 3.2.3. For CGRAs in which the hardware does not support parallel threads, the compiler can still merge the DDGs of multiple loops, and schedule them together, onto subpartitions of the CGRA [80]. That way, software-controlled multi-threading can still be achieved.

The aforementioned algorithms have been extended to not only consider the costs of utilized resources inside the CGRA during scheduling, but to also consider bank conflicts that may occur because of multiple memory accesses being scheduled in the same cycle [49, 50].

Many other CGRA compiler techniques have been proposed, most of which are restricted to specific architectures. Static reconfigurable architectures like RaPiD

and PACT have been targeted by compiler algorithms [16, 26, 114] based on placement and routing techniques that also map DDGs on RRGs. These techniques support subsets of the C programming language (no pointers, no structs, ...) and require the use of special C functions to program the IO in the loop bodies to be mapped onto the CGRA. The latter requirement follows from the specific IO support in the architectures and the modeling thereof in the RRGs.

For the MorphoSys architecture, with its emphasis on SIMD across ISs, compiler techniques have been developed for the SA-C language [111]. In this language the supported types of available parallelism are specified by means of loop language constructs. These constructs are translated into control code for the CGRA, which are mapped onto the ISs together with the DDGs of the loop bodies.

CGRA code generation techniques based on integer-linear programming have been proposed for the several architectures, both for spatial [2] and for temporal mapping [56, 127]. Basically, the ILP formulation consists of all the requirements or constraints that must be met by a valid schedule. This formulation is built from a DDG and a hardware description, and can hence be used to compile many source languages. It is unclear, however, to what extent the ILP formulation and its solution rely on specific architecture features, and hence to which extent it would be possible to retarget the ILP-formulation to different CGRA designs. A similar situation occurs for the constraint-based compilation method developed for the Silicon Hive architecture template [101], of which no detailed information is public. Furthermore, ILP-based compilation is known to be unreasonably slow. So in practice it can only be used for small loop kernels.

Code generation techniques for CGRAs based on instruction-selection pattern matching and list-scheduling techniques have also been proposed [30, 31]. It is unclear to what extent these techniques rely on a specific architecture because we know of no trial to use them for different CGRAs, but these techniques seem to rely heavily on the existence of a single shared-bus that connects ISs as depicted in Fig. 7c. Similarly, the static reconfiguration code generation technique by Lee et al. relies on CGRA rows consisting of identical ISs [58]. Because of this assumption, a two-step code generation approach can be used in which individual placements within rows are neglected in the first step, and only taken care of in the second step. The first step then instead focuses on optimizing the memory traffic.

Finally, compilation techniques have been developed that are really specialized for the TRIPS array layout and for its out-of-order execution [20].

4 Case Study: ADRES

This section presents a case study on one specific CGRA design template. The purpose of this study is to illustrate that it is non-trivial to compile and optimize code for CGRA targets, and to illustrate that within a design template, there is a need for hardware design exploration. This illustrates how both hardware and software

designers targeting CGRAs need a deep understanding of the interaction between the architecture features and the used compiler techniques.

ADRES [7, 11–13, 23, 24, 71, 73–75] is an architecture design template from which dynamically reconfigurable, statically scheduled CGRAs can be instantiated. In each instance, an ADRES CGRA is coupled tightly to a VLIW processor. This processor shares data and predicate RFs with the CGRA, as well as memory ports to a multi-banked scratch-pad memory as described in Sect. 3.1. The compiler-supported ISA of the design template provides instructions that are typically found in a load/store VLIW or RISC architecture, including arithmetic operations, logic operations, load/store operations, and predicate computing instructions. Additional domain-specific instructions, such as SIMD operations, are supported in the programming tools by means of intrinsics [102]. Local rotating and non-rotating, shared and private local RFs can be added to the CGRA as described in the previous sections, and connected through an interconnect consisting of muxes, buses and point-to-point connections that are specified completely by the designer. Thus, the ADRES architecture template is very flexible: it offers a high degree of design freedom, and it can be used to accelerate a wide range of loops.

4.1 Mapping Loops on ADRES CGRAs

The first part of this case study concerns the mapping of loops onto ADRES CGRAs, which are one of the most flexible CGRAs supporting a wide range of loops. This study illustrates that many loop transformations need to be applied carefully before mapping code onto ADRES CGRAs. We discuss the most important compiler transformations and, lacking a full-fledged loop-optimizing compiler, manual loop transformations that need to be applied to source code in order to obtain high performance and high efficiency. For other, less flexible CGRAs, the need for such transformations will even be higher because there will be more constraints on the loops to be mapped in the first place. Hence many of the discussed issues not only apply to ADRES CGRAs, but also to other CGRA architectures. We will conclude from this study that programming CGRAs with the existing compiler technology is not compatible with high programmer productivity.

4.1.1 Modulo Scheduling Algorithms for CGRAs

To exploit ILP in inner loops on VLIW architectures, compilers typically apply software pipelining by means of modulo scheduling [54, 93]. This is no different for ADRES CGRAs. In this section, we will not discuss the inner working of modulo scheduling algorithms. What we do discuss, are the consequences of using that technique for programming ADRES CGRAs.

After a loop has been modulo-scheduled, it consists of three phases: the prologue, the kernel and the epilogue. During the prologue, stages of the software-pipelined

loop gradually become active. Then the loop executes the kernel in a steady-state mode in which all software pipeline stages are active, and afterwards the stages are gradually disabled during the epilogue. In the steady-state mode, a new iteration is started after every II cycles, which stands for Initiation Interval. Fundamentally, every software pipeline stage is II cycles long. The total cycle count of a loop with *iter* iterations that is scheduled over ps software pipeline stages is then given by

$$cycles_{prologue} + II \cdot (iter - (ps - 1)) + cycles_{epilogue}. \tag{2}$$

In this formula, we neglect processor stalls because of, e.g., memory access conflicts or cache misses.

For loops with a high number of iterations, the term $II \cdot iter$ dominates this cycle count, and that is why modulo scheduling algorithms try to minimize II, thus increasing the IPC terms in Eq. (1).

The minimal II that modulo scheduling algorithms can reach is bound by $minII = \max(RecMII, ResMII)$. The first term, called resource-minimal II ($ResMII$) is determined by the resources required by a loop and by the resources provided by the architecture. For example, if a loop body contains nine multiplications, and there are only two ISs that can execute multiplications, then at least $\lceil 9/2 \rceil = 5$ cycles will be needed per iteration. The second term, called recurrence-minimal II ($RecMII$) depends on recurrent data dependencies in a loop and on instruction latencies. Fundamentally, if an iteration of a loop depends on the previous iteration through a dependency chain with accumulated latency $RecMII$, it is impossible to start that iteration before at least $RecMII$ cycles of the previous iteration have been executed.

The next section uses this knowledge to apply transformations that optimize performance according to Eq. (1). To do so successfully, it is important to know that ADRES CGRAs support only one thread, for which the processor has to switch from a non-CGRA operating mode to CGRA mode and back for each inner loop. So besides minimizing the cycle count of Eq. (2) to obtain higher IPCs in Eq. (1), it is also important to consider the terms $t_{p \rightarrow p+1}$ in Eq. (1).

4.1.2 Loop Transformations

Loop Unrolling

Loop unrolling and the induction variable optimizations that it enables can be used to minimize the number of iterations of a loop. When a loop body is unrolled x times, $iter$ decreases with a factor x, and $ResMII$ typically grows with a factor slightly less than x because of the induction variable optimizations and because of the ceiling operation in the computation of $ResMII$. By contrast, $RecMII$ typically remains unchanged or increases only a little bit as a result of the induction variable optimizations that are enabled after loop unrolling.

In resource-bound loops, $ResMII > RecMII$. Unrolling will then typically have little impact on the dominating term $II \cdot iter$ in Eq. (2). However, the prologue and the epilogue will typically become longer because of loop unrolling. Moreover, an unrolled loop will consume more space in the instruction memory, which might also have a negative impact on the total execution time of the whole application. So in general, unrolling resource-bound loops is unlikely to be very effective.

In recurrence-bound loops, $RecMII \cdot iter > ResMII \cdot iter$. The right hand side of this inequality will not increase by unrolling, while the left hand side will be divided by the unrolling factor x. As this improvement typically compensates for the longer prologue and epilogue, we can conclude that unrolling can be an effective optimization technique for recurrence-bound loops if the recurrences can be optimized with induction variable optimizations. This is no different for CGRAs than it is for VLIWs. However, for CGRAs with their larger number of ISs, it is more important because more loops are recurrence-bound.

Loop Fusion, Loop Interchange, Loop Combination and Data Context Switching

Fusing adjacent loops with the same number of iterations into one loop can also be useful, because fusing multiple recurrence-bound loops can result in one resource-bound loop, which will result in a lower overall execution time. Furthermore, less switching between operating modes takes place with fused loops, and hence the terms $t_{p \rightarrow p+1}$ are minimized. Furthermore, less prologues and epilogues need to be executed, which might also improve performance. This improvement will usually be limited, however, because the fused prologues and epilogues will rarely be much shorter than the sum of the original ones. Moreover, loop fusion does result in a loop that is bigger than any of the original loops, so it can only be applied if the configuration memory is big enough to fit the fused loop. If this is the case, less loop configurations need to be stored and possibly reloaded into the memory.

Interchanging an inner and outer loop serves largely the same purpose as loop fusion. As loop interchange does not necessarily result in larger prologues and epilogues, it can be even more useful, as can be the combining of nested loops into a single loop. Data-context switching [10] is a very similar technique that serves the same purpose. That technique has been used by Lee et al. for statically reconfigurable CGRAs as well [58], and in fact most of the loop transformations mentioned in this section can be used to target such CGRAs, as well as any other type of CGRA.

Live-In Variables

In our experience, there is only one caveat with the above transformations. The reason to be careful when applying them is that they can increase the number of live-in variables. A live-in variable is a variable that gets assigned a value before the

loop, which is consequently used in the loop. Live-in variables can be manifest in the original source code, but they can also result from compiler optimizations that are enabled by the above loop transformations, such as induction variable optimizations and loop-invariant code motion. When the number of live-in variables increases, more data needs to be passed from the non-loop code to the loop code, which might have a negative effect on $t_{p \rightarrow p+1}$. The existence and the scale of this effect will usually depend on the hardware mechanism that couples the CGRA accelerator to the main core. Possible such mechanisms are discussed in Sect. 3.1. In tightly-coupled designs like that of ADRES or Silicon Hive, passing a limited amount of values from the main CPU mode to the CGRA mode does not involve any overhead: the values are already present in the shared RF. However, if their number grows too big, there will not be enough room in the shared RF, which will result in much less efficient passing of data through memory. We have experienced this several times with loops in multimedia and SDR applications that were mapped onto our ADRES designs. So, even for tightly-coupled CGRA designs, the above loop transformations and the enabled optimizations need to be applied with great care.

Predication

The "basic" modulo scheduling techniques for CGRAs [24, 26, 29, 73, 75, 78, 81, 82, 107] only schedule loops that are free of control flow transfers. Hence any loop body that contains conditional statements first needs to be if-converted into hyperblocks by means of predication [70]. For this reason, many CGRAs, including ADRES CGRAs, support predication.

Hyperblock formation can result in very inefficient code if a loop body contains code paths that are executed rarely. All those paths contribute to $ResMII$ and potentially to $RecMII$. Hence even paths that get executed very infrequently can slow down a whole modulo-scheduled loop. Such loops can be detected with profiling, and if the data dependencies allow this, it can be useful to split these loops into multiple loops. For example, a first loop can contain the code of the frequently executed paths only, with a lower II than the original loop. If it turns out during the execution of this loop that in some iteration the infrequently executed code needs to be executed, the first loop is exited, and for the remaining iterations a second loop is entered that includes both the frequently and the infrequently executed code paths.

Alternatively, for some loops it is beneficial to have a so-called inspector loop with very small II to perform only the checks for all iterations. If none of the checks are positive, a second so-called executor loop is executed that includes all the computations except the checks and the infrequently executed paths. If some checks were positive, the original loop is executed.

One caveat with this loop splitting is that it causes code size expansion in the CGRA instruction memories. For power consumption reasons, these memories are kept as small as possible. This means that the local improvements obtained with the loop splitting need to be balanced with the total code size of all loops that need to share these memories.

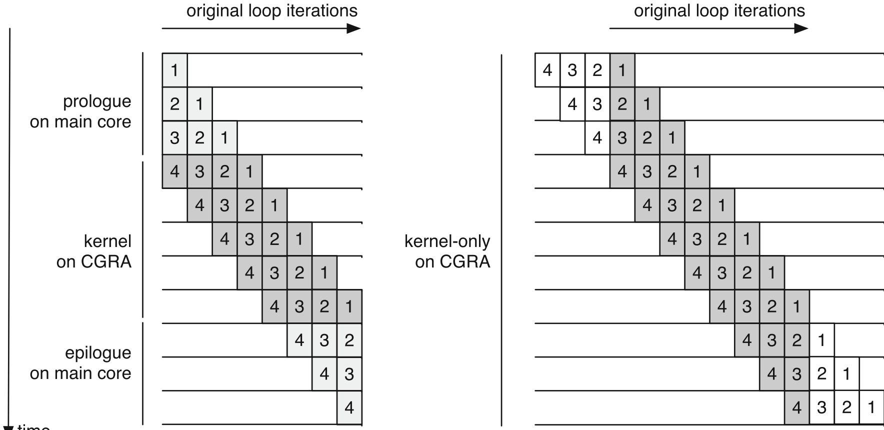


Fig. 8 On the left a traditional modulo-scheduled loop, on the right a kernel-only one. Each numbered box denotes one of four software pipeline stages, and each row denotes the concurrent execution of different stages of different iterations. Grayed boxes denote stages that actually get executed. On the left, the dark grayed boxes get executed on the CGRA accelerator, in which exactly the same code is executed every II cycles. The light grayed boxes are pipeline stages that get executed outside of the loop, in separate code that runs on the main processor. On the right, kernel-only code is shown. Again, the dark grey boxes are executed on the CGRA accelerator. So are the white boxes, but these get deactivated during the prologue and epilogue by means of predication

Kernel-Only Loops

Predication can also be used to generate so-called kernel-only loop code. This is loop code that does not have separate prologue and epilogue code fragments. Instead the prologues and epilogues are included in the kernel itself, where predication is now used to guard whole software pipeline stages and to ensure that only the appropriate software pipeline stages are activated at each point in time. A traditional loop with a separate prologue and epilogue is compared to a kernel-only loop in Fig. 8. Three observations need to be made here.

The first observation is that kernel-only code is usually faster because the pipeline stages of the prologue and epilogue now get executed on the CGRA accelerator, which typically can do so at much higher IPCs than the main core. This is a major difference between (ADRES) CGRAs and VLIWs. On the latter, kernel-only loops are much less useful because all code runs on the same number of ISs anyway.

Secondly, while kernel-only code will be faster on CGRAs, more time is spent in the CGRA mode, as can be seen in Fig. 8. During the epilogue and prologue, the whole CGRA is active and thus consuming energy, but many ISs are not performing useful computations because they execute operations from inactive pipeline stages. Thus, kernel-only is not necessarily optimal in terms of energy consumption.

The third observation is that for loops where predication is used heavily to create hyperblocks, the use of predicates to support kernel-only code might over-stress

Table 1 Main differences between two studied ADRES CGRAs

	multimedia CGRA	**SDR CGRA**
# issue slots (FUs)	4x4	4x4
# load/store units	4	4
ld/st/mul latency	6/6/2 cycles	7/7/3 cycles
# local data RFs	12 (8 single-ported) of size 8	12 (8 single-ported) of size 4
data width	32	64
config. word width	896 bits	736 bits
ISA extensions	2-way SIMD, clipping, min/max	4-way SIMD, saturating arithm.
interconnect	Nearest Neighbor (NN) + 8 predicate buses	NN + next-hop + 8 data buses
pipelining	RF mux mux FU	RF mux mux FU
power, clock, and area	91 mW at 300 MHz for 4mm^2	310mW at 400 MHz for 5.79mm^2

Power, clock and area include the CGRA and its configuration memory, the VLIW processor for non-loop code, including its 32K L1 I-cache, and the 32K 4-bank L1 data memory. These numbers are gate-level estimates

the predication support of the CGRA. In domains such as SDR, where the loops typically have no or very little conditional statements, this poses no problems. For applications that feature more complex loops, such as in many multimedia applications, this might create a bottleneck even when predicate speculation [97] is used. This is where the ADRES template proves to be very useful, as it allowed us to instantiate specialized CGRAs with varying predicate data paths, as can be seen in Table 1.

4.1.3 Data Flow Manipulations

The need for fine-tuning source code is well known in the embedded world. In practice, each compiler can handle some loop forms better than other forms. So when one is using a specific compiler for some specific VLIW architecture, it can be very beneficial to bring loops in the appropriate shape or form. This is no different when one is programming for CGRAs, including ADRES CGRAs.

Apart from the above transformations that relate to the modulo scheduling of loops, there are important transformations that can increase the "data flow" character of a loop, and thus contribute to the efficiency of a loop. Three C implementations of a Finite Impulse Response (FIR) filter in Fig. 9 provide an excellent example.

Figure 9a depicts a FIR implementation that is efficient for architectures with few registers. For architectures with more registers, the implementation depicted in Fig. 9b will usually be more efficient, as many memory accesses have been

a
```
const short c[15] = {-32, ..., 1216};
for (i = 0; i < nr; i++) {
  for(value = 0, j = 0; j < 15; j++)
    value += x[i+j]*c[j];
  r[i] = value;
}
```

b
```
const short c00 = -32, ..., c14 = 1216;
for (i = 0; i < nr; i++)
  r[i] = x[i+0]*c00 + x[i+1]*c01 + ... + x[i+14]*c14;
```

c
```
int i, value, d0, ..., d14;
const short c00 = -32, ..., c14 = 1216;
for (i = 0; i < nr+15; i++) {
  d0 = d1;  d1 = d2;  ... ; d13 = d14; d14 = x[i];
  value = c00 * d0 +  c01 * d1 + ... + c14 * d14;
  if (i >= 14) r[i - 14 ] = value;
}
```

Fig. 9 Three C versions of a FIR filter. (**a**) Original 15-tap FIR filter, (**b**) filter after loop unrolling, with hard-coded constants, (**c**) after redundant memory accesses are eliminated

Table 2 Number of execution cycles and memory accesses (obtained through simulation) for the FIR-filter versions compiled for the multimedia CGRA, and for the TI C64+ DSP

	Cycle count		Memory accesses	
Program	CGRA	TI C64+	CGRA	TI C64+
FIR (a)	11,828	1054	6221	1618
FIR (b)	1247	1638	3203	2799
FIR (c)	664	10,062	422	416

eliminated. Finally, the equivalent code in Fig. 9c contains only one load per outer loop iteration. To remove the redundant memory accesses, a lot of temporary variables had to be inserted, together with a lot of copy operations that implement a delay line. On regular VLIW architectures, this version would result in high register pressure and many copy operations to implement the data flow of those copy operations. Table 2 presents the compilation results for a 16-issue CGRA and for an 8-issue clustered TI C64+ VLIW. From the results, it is clear that the TI compiler could not handle the latter code version: its software-pipelining fails completely due to the high register pressure. When comparing the minimal cycle times obtained for the TI C64+ with those obtained for the CGRA, please note that the TI compiler applied SIMDization as much as it could, which is fairly orthogonal to scheduling and register allocation, but which the experimental CGRA compiler used for this experiment did not yet perform. By contrast, the CGRA compiler could optimize the code of Fig. 9c by routing the data of the copy operations over direct connections

between the CGRA ISs. As a result, the CGRA implementation becomes both fast and power-efficient at the same time.

This is a clear illustration of the fact that, lacking fully automated compiler optimizations, heavy performance-tuning of the source code can be necessary. The fact that writing efficient source code requires a deep understanding of the compiler internals and of the underlying architecture, and the fact that it frequently includes experimentation with various loop shapes, severely limits the programming productivity. This has to be considered a severe drawback of CGRAs architectures.

Moreover, as the FIR filter shows, the optimal source code for a CGRA target can be radically different than that for, e.g., a VLIW target. Consequently, the cost of porting code from other targets to CGRAs or vice versa, or of maintaining code versions for different targets (such as the main processor and the CGRA accelerator), can be high. This puts an additional limitation on programmer productivity.

4.2 ADRES Design Space Exploration

In this part of our case study, we discuss the importance and the opportunities for DSE within the ADRES template. First, we discuss some concrete ADRES instances that have been used for extensive experimentation, including the fabrication of working silicon samples. These examples demonstrate that very power-efficient CGRAs can be designed for specific application domains.

Afterwards, we will show some examples of DSE results with respect to some of the specific design options that were discussed in Sect. 3.

4.2.1 Example ADRES Instances

During the development of the ADRES tool chain and design, two main ADRES instances have been worked out. One was designed for multimedia applications [7, 71] and one for SDR baseband processing [11, 12]. Their main differences are presented in Table 1. Both architectures have a 64-entry data RF (half rotating, half non-rotating) that is shared with a unified three-issue VLIW processor that executes non-loop code. Thus this shared RF has six read ports and three write ports. Both CGRAs feature 16 FUs, of which four can access the memory (that consists of four single-ported banks) through a queue mechanism that can resolve bank conflicts. Most operations have latency one, with the exception of loads, stores, and multiplications. One important difference between the two CGRAs relates to their pipeline schemes, as depicted for a single IS (local RF and FU) in Table 1. As the local RFs are only buffered at their input, pipelining registers need to be inserted in the paths to and from the FUs in order to obtain the desired frequency targets as indicated in the table. The pipeline latches shown in Table 1 hence directly contribute in the maximization of the factor f_p in Eq. (1). Because the instruction sets and the target frequencies are different in both application domains, the SDR

CGRA has one more pipeline register than the multimedia CGRA, and they are located at different places in the design.

Traditionally, in VLIWs or in out-of-order superscalar processors, deeper pipelining results in higher frequencies but also in lower IPCs because of larger branch misprediction penalties. Following Eq. (1), this can result in lower performance. In CGRAs, however, this is not necessarily the case, as explained in Sect. 3.3.1. To illustrate this, Table 3 includes IPCs obtained when generating code for both CGRAs with and without the pipelining latches.

The benchmarks mapped onto the multimedia ADRES CGRA are a H.264AVC video decoder, a wavelet-based video decoder, an MPEG4 video coder, a black-and-white TIFF image filter, and a SHA-2 encryption algorithm. For each application at most the 10 hottest inner loops are included in the table. For the SDR ADRES CGRA, we selected two baseband modem benchmarks: one WLAN MIMO Channel Estimation and one that implements the remainder of a WLAN SISO receiver. All applications are implemented in standard ANSI C using all language features such as pointers, structures, different loop constructs (while, for, do-while), but not using dynamic memory management functions like `malloc` or `free`.

The general conclusions to be taken from the mapping results in Table 3 are as follows. (1) Very high IPCs are obtained at low power consumption levels of 91 and 310 mW and at relatively high frequencies of 300 and 400 MHz, given the standard cell 90 nm design. (2) Pipelining seems to be bad for performance only where the initiation interval is bound by $RecMII$, which changes with pipelining. (3) In some cases pipelining even improves the IPC.

Synthesizable VHDL is generated for both processors by a VHDL generator that generates VHDL code starting from the same XML architecture specification used to retarget the ANSI C compiler to different CGRA instances. A TSMC 90 nm standard cell GP CMOS (i.e. the General-Purpose technology version that is optimized for performance and active power, not for leakage power) technology was used to obtain the gate-level post-layout estimates for frequency, power and area in Table 1. More detailed results of these experiments are available in the literature for this SDR ADRES instance [11, 12], as well as for the multimedia instance [7, 71]. The SDR ADRES instance has also been produced in silicon in samples of a full SoC SDR chip [25]. The two ADRES cores on this SoC proved to be fully functional at 400 MHz, and the power consumption estimates have been validated.

One of the most interesting results is depicted in Fig. 10, which displays the average power consumption distribution over the ADRES SDR CGRA when the CGRA mode is active in the above SDR applications. Compared to VLIW processor designs, a much larger fraction of the power is consumed in the interconnects and in the FUs, while the configuration memory (which corresponds to an L1 VLIW instruction cache), the RFs and the data memory consume relatively little energy. This is particularly the case for the local RFs. This clearly illustrates that by focusing on regular loops and their specific properties, CGRAs can achieve higher performance and a higher power-efficiency than VLIWs. On the CGRA, most of the power is spent in the FUs and in the interconnects, i.e., on the actual computations and on the transfers of values from computation to computation. The latter two

Table 3 Results for the benchmark loops

					Pipelined			Non-pipelined		
Benchmark	CGRA	Loop	#ops	ResMII	RecMII	II	IPC	RecMII	II	IPC
AVC decoder	Multimedia	MBFilter1	70	5	2	6	11.7	1	6	11.7
		MBFilter2	89	6	7	9	9.9	6	8	11.1
		MBFilter3	40	3	3	4	10.0	2	3	13.3
		MBFilter4	105	7	2	9	11.7	1	9	11.7
		MotionComp	109	7	3	10	10.9	2	10	10.9
		FindFrameEnd	27	4	7	7	3.9	6	6	4.5
		IDCT1	60	4	2	5	12.0	1	5	12.0
		MBFilter5	87	6	3	7	12.4	2	7	12.4
		Memset	10	2	2	2	5.0	1	2	5.0
		IDCT2	38	3	2	3	12.7	1	3	12.7
		Average					10.0			10.5
Wavelet	Multimedia	Forward1	67	5	5	6	11.2	5	5	13.4
		Forward2	77	5	5	6	12.8	5	6	12.8
		Reverse1	73	5	2	6	12.2	1	6	12.2
		Reverse2	37	3	2	3	12.3	1	3	12.3
		Average					12.1			12.7
MPEG-4 encoder	Multimedia	MotionEst1	75	5	2	6	12.5	1	6	12.5
		MotionEst2	72	5	3	6	12.0	2	6	12.0
		TextureCod1	73	5	7	7	10.4	6	6	12.2
		CalcMBSAD	60	4	2	5	12.0	1	5	12.0
		TextureCod2	9	1	2	2	4.5	1	2	4.5
		TextureCod3	91	6	2	7	13.0	1	7	13.0
		TextureCod4	91	6	2	7	13.0	1	7	13.0
		TextureCod5	82	6	2	6	13.7	1	6	13.7
		TextureCod6	91	6	2	7	13.0	1	7	13.0
		MotionEst3	52	4	3	4	13.0	2	5	10.4
		Average					11.7			11.6
Tiff2BW	Multimedia	Main loop	35	3	2	3	11.7	1	3	11.7
SHA-2	Multimedia	Main loop	111	7	8	9	12.3	8	9	12.3
MIMO	SDR	Channel2	166	11	3	14	11.9	1	14	10.4
		Channel1	83	6	3	8	10.4	1	8	10.7
		SNR	75	5	4	6	12.5	2	6	12.5
		Average					11.6			11.2
WLAN	SDR	DemapQAM64	55	4	3	6	9.2	1	6	9.2
		64-point FFT	123	8	4	10	12.3	2	12	10.3
		Radix8 FFT	122	8	3	10	12.2	1	12	10.2
		Compensate	54	4	4	5	10.8	2	5	10.8
		DataShuffle	153	14	3	14	10.9	1	16	9.6
		Average					11.1			10.0

First, the target-version-independent number of operations (#ops) and the ResMII. Then for each target version the RecMII, the actually achieved II and IPC (counting SIMD operations as only one operation), and the compile time

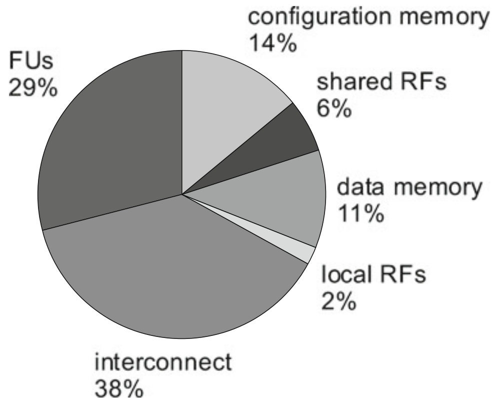


Fig. 10 Average power consumption distribution of the ADRES SDR CGRA in CGRA mode

aspects are really the fundamental parts of the computation to be performed, unlike the fetching of data or the fetching of code, which are merely side-effects of the fact that processors consist of control paths, data paths, and memories.

4.2.2 Design Space Exploration Example

Many DSEs have been performed within the ADRES template [7, 13, 18, 55, 71, 77]. We present one experimental result [55] here, not to present absolute numbers but to demonstrate the large impact on performance and on energy consumption that some design choices can have. In this experiment, a number of different interconnects have been explored for four microbenchmarks (each consisting of several inner loops): a MIMO SDR channel estimation, a Viterbi decoder, an Advanced Video Codec (AVC) motion estimation, and an AVC half-pixel interpolation filter. All of them have been compiled with the DRESC compiler for different architectures of which the interconnects are combinations of the four basic interconnects of Fig. 6, in which distributed RFs have been omitted for the sake of clarity.

Figure 11 depicts the relative performance and (estimated) energy consumption for different combinations of these basic interconnects. The names of the different architectures indicate which basic interconnects are included in its interconnect. For example, the architecture `b_nn_ex` includes the buses, nearest neighbor interconnects and extra connections to the shared RF. The lines connecting architectures in the charts of Fig. 11 connect the architectures on the Pareto fronts: these are the architectures that have an optimal combination of cycle count and energy consumption. Depending on the trade-off made by a designer between performance and energy consumption, he will select one architecture on that Pareto front.

The lesson to learn from these Pareto fronts is that relatively small architectural changes, in this case involving only the interconnect but not the ISs or the distributed RFs, can span a wide range of architectures in terms of performance and energy-efficiency. When designing a new CGRA or choosing for an existing one, it is hence

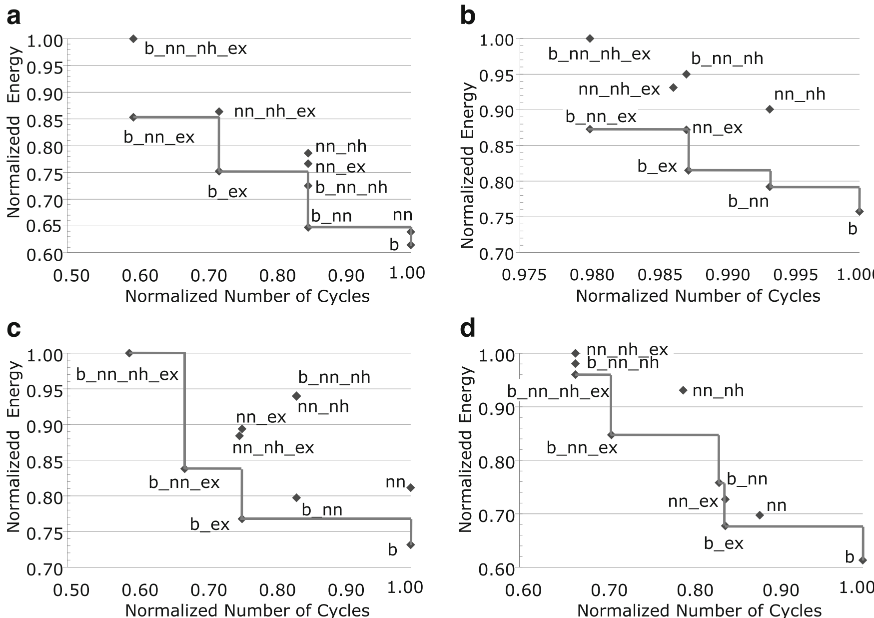


Fig. 11 DSE results for four microbenchmarks on 4 × 4 CGRAs with fixed ISs and fixed RFs, but with varying interconnects. (**a**) MIMO, (**b**) AVC interpolation, (**c**) Viterbi, (**d**) AVC motion estimation

absolutely necessary to perform a good DSE that covers ISA, ISs, interconnect and RFs. Because of the large design space, this is far from trivial.

5 Conclusions

This chapter on CGRA architectures presented a discussion of the CGRA processor design space as an accelerator for inner loops of DSP-like applications such as software-defined radios and multimedia processing. A range of options for many design features and design parameters has been related to power consumption, performance, and flexibility. In a use case, the need for design space exploration and for advanced compiler support and manual high-level code tuning have been demonstrated. The above discussions and demonstration support the following main conclusions. Firstly, CGRAs can provide an excellent alternative for VLIWs, providing better performance and better energy efficiency. Secondly, design space exploration is needed to achieve those goals. Finally, existing compiler support needs to be improved, and until that happens, programmers need to have a deep understanding of the targeted CGRA architectures and their compilers in order to manually tune their source code. This can significantly limit programmer productivity.

6 Further Reading

For further reading, the historic development of the ADRES architecture is interesting, from the first academic conception of the architecture and its initial compiler support [73–75], over the first fabricated prototypes [12, 25], to their commercial derivatives [44]. The historic development of appropriate compiler models [24] and scheduling techniques [78, 81, 82, 107] to achieve both high code quality and fast compilation is interesting as well.

Some of the more interesting recent research directions include power optimization by means of adaptive and multiple V_{dd}'s [33, 115, 120], architectural and compiler support for nested loops [57, 121–123], more dynamic control [126], and support for thread-level parallelism [80, 83, 110].

For pointers for further reading on other specific design aspects of CGRAs, we refer to the corresponding sections in this chapter, which include plenty of references.

References

1. Abnous, A., Christensen, C., Gray, J., Lenell, J., Naylor, A., Bagherzadeh, N.: Design and implementation of the "Tiny RISC" microprocessor. Microprocessors & Microsystems **16**(4), 187–193 (1992)
2. Ahn, M., Yoon, J.W., Paek, Y., Kim, Y., Kiemb, M., Choi, K.: A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In: DATE '06: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 363–368 (2006)
3. Ansaloni, G., Bonzini, P., Pozzi, L.: EGRA: A coarse grained reconfigurable architectural template. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **19**(6), 1062–1074 (2011)
4. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Comput. Surv. **26**(4), 345–420 (1994)
5. Barua, R.: Maps: a compiler-managed memory system for software-exposed architectures. Ph.D. thesis, Massachusetts Institute of Technology (2000)
6. Benabderrahmane, M.W., Pouchet, L.N., Cohen, A., Bastoul, C.: The polyhedral model is more widely applicable than you think. In: Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction, CC'10/ETAPS'10, pp. 283–303. Springer-Verlag, Berlin, Heidelberg (2010)
7. Berekovic, M., Kanstein, A., Mei, B., De Sutter, B.: Mapping of nomadic multimedia applications on the ADRES reconfigurable array processor. Microprocessors & Microsystems **33**(4), 290–294 (2009)
8. van Berkel, k., Heinle F. amd Meuwissen, P., Moerman, K., Weiss, M.: Vector processing as an enabler for software-defined radio in handheld devices. EURASIP Journal on Applied Signal Processing **2005**(16), 2613–2625 (2005)
9. Betz, V., Rose, J., Marguardt, A.: Architecture and CAD for Deep-Submicron FPGAs. Kluwer Academic Publishers (1999)
10. Bondalapati, K.: Parallelizing DSP nested loops on reconfigurable architectures using data context switching. In: DAC '01: Proceedings of the 38th annual Design Automation Conference, pp. 273–276 (2001)

11. Bougard, B., De Sutter, B., Rabou, S., Novo, D., Allam, O., Dupont, S., Van der Perre, L.: A coarse-grained array based baseband processor for 100Mbps+ software defined radio. In: DATE '08: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 716–721 (2008)
12. Bougard, B., De Sutter, B., Verkest, D., Van der Perre, L., Lauwereins, R.: A coarse-grained array accelerator for software-defined radio baseband processing. IEEE Micro **28**(4), 41–50 (2008). http://doi.ieeecomputersociety.org/10.1109/MM.2008.49
13. Bouwens, F., Berekovic, M., Gaydadjiev, G., De Sutter, B.: Architecture enhancements for the ADRES coarse-grained reconfigurable array. In: HiPEAC '08: Proceedings of the International Conference on High-Performance Embedded Architectures and Compilers, pp. 66–81 (2008)
14. Burns, G., Gruijters, P.: Flexibility tradeoffs in SoC design for low-cost SDR. Proceedings of SDR Forum Technical Conference (2003)
15. Burns, G., Gruijters, P., Huiskens, J., van Wel, A.: Reconfigurable accelerators enabling efficient SDR for low cost consumer devices. Proceedings of SDR Forum Technical Conference (2003)
16. Cardoso, J.M.P., Weinhardt, M.: XPP-VC: A C compiler with temporal partitioning for the PACT-XPP architecture. In: FPL '02: Proceedings of the 12th International Conference on Field-Programmable Logic and Applications, pp. 864–874 (2002)
17. Cervero, T.: Analysis, implementation and architectural exploration of the H.264/AVC decoder onto a reconfigurable architecture. Master's thesis, Universidad de Los Palmas de Gran Canaria (2007)
18. Cervero, T., Kanstein, A., López, S., De Sutter, B., Sarmiento, R., Mignolet, J.Y.: Architectural exploration of the H.264/AVC decoder onto a coarse-grain reconfigurable architecture. In: Proceedings of the International Conference on Design of Circuits and Integrated Systems (2008)
19. Chen, L., Mitra, T.: Graph minor approach for application mapping on CGRAs. ACM Trans. on Reconf. Technol. and Systems **7**(3), 21 (2014)
20. Coons, K.E., Chen, X., Burger, D., McKinley, K.S., Kushwaha, S.K.: A spatial path scheduling algorithm for EDGE architectures. In: ASPLOS '06: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 129–148 (2006)
21. Corporaal, H.: Microprocessor Architectures from VLIW to TTA. John Wiley (1998)
22. Cronquist, D., Franklin, P., Fisher, C., Figueroa, M., Ebeling, C.: Architecture design of reconfigurable pipelined datapaths. In: Proceedings of the Twentieth Anniversary Conference on Advanced Research in VLSI (1999)
23. De Sutter, B., Allam, O., Raghavan, P., Vandebriel, R., Cappelle, H., Vander Aa, T., Mei, B.: An efficient memory organization for high-ILP inner modem baseband SDR processors. Journal of Signal Processing Systems **61**(2), 157–179 (2010)
24. De Sutter, B., Coene, P., Vander Aa, T., Mei, B.: Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. In: LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 151–160 (2008)
25. Derudder, V., Bougard, B., Couvreur, A., Dewilde, A., Dupont, S., Folens, L., Hollevoet, L., Naessens, F., Novo, D., Raghavan, P., Schuster, T., Stinkens, K., Weijers, J.W., Van der Perre, L.: A 200Mbps+ 2.14nJ/b digital baseband multi processor system-on-chip for SDRs. In: Proceedings of the Symposium on VLSI Systems, pp. 292–293 (2009)
26. Ebeling, C.: Compiling for coarse-grained adaptable architectures. Tech. Rep. UW-CSE-02-06-01, University of Washington (2002)
27. Ebeling, C.: The general RaPiD architecture description. Tech. Rep. UW-CSE-02-06-02, University of Washington (2002)

28. Fisher, J., Faraboschi, P., Young, C.: Embedded Computing, A VLIW Approach to Architecture, Compilers and Tools. Morgan Kaufmann (2005)
29. Friedman, S., Carroll, A., Van Essen, B., Ylvisaker, B., Ebeling, C., Hauck, S.: SPR: an architecture-adaptive CGRA mapping tool. In: FPGA '09: Proceeding of the ACM/SIGDA International symposium on Field Programmable Gate Arrays, pp. 191–200. ACM, New York, NY, USA (2009)
30. Galanis, M.D., Milidonis, A., Theodoridis, G., Soudris, D., Goutis, C.E.: A method for partitioning applications in hybrid reconfigurable architectures. Design Automation for Embedded Systems **10**(1), 27–47 (2006)
31. Galanis, M.D., Theodoridis, G., Tragoudas, S., Goutis, C.E.: A reconfigurable coarse-grain data-path for accelerating computational intensive kernels. Journal of Circuits, Systems and Computers pp. 877–893 (2005)
32. Gebhart, M., Maher, B.A., Coons, K.E., Diamond, J., Gratz, P., Marino, M., Ranganathan, N., Robatmili, B., Smith, A., Burrill, J., Keckler, S.W., Burger, D., McKinley, K.S.: An evaluation of the TRIPS computer system. In: ASPLOS '09: Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 1–12 (2009)
33. Gu, J., Yin, S., Liu, L., Wei, S.: Energy-aware loops mapping on multi-v_{dd} CGRAs without performance degradation. In: 22nd Asia and South Pacific Design Automation Conference, ASP-DAC 2017, Chiba, Japan, January 16–19, 2017, pp. 312–317 (2017)
34. Hamzeh, M., Shrivastava, A., Vrudhula, S.: EPIMap: using epimorphism to map applications on CGRAs. In: Proc. 49th Annual Design Automation Conf., pp. 1284–1291 (2012)
35. Hamzeh, M., Shrivastava, A., Vrudhula, S.B.K.: REGIMap: register-aware application mapping on coarse-grained reconfigurable architectures (CGRAs). In: Proc. Annual Design Automation Conf., pp. 1–10 (2013)
36. Hamzeh, M., Shrivastava, A., Vrudhula, S.B.K.: Branch-aware loop mapping on CGRAs. In: The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1–5, 2014, pp. 107:1–107:6 (2014)
37. Hartenstein, R., Herz, M., Hoffmann, T., Nageldinger, U.: Mapping applications onto reconfigurable KressArrays. In: Proceedings of the 9th International Workshop on Field Programmable Logic and Applications (1999)
38. Hartenstein, R., Herz, M., Hoffmann, T., Nageldinger, U.: Generation of design suggestions for coarse-grain reconfigurable architectures. In: FPL '00: Proceedings of the 10th International Workshop on Field Programmable Logic and Applications (2000)
39. Hartenstein, R., Hoffmann, T., Nageldinger, U.: Design-space exploration of low power coarse grained reconfigurable datapath array architectures. In: Proceedings of the International Workshop - Power and Timing Modeling, Optimization and Simulation (2000)
40. Hartmann, M., Pantazis, V., Vander Aa, T., Berekovic, M., Hochberger, C., De Sutter, B.: Still image processing on coarse-grained reconfigurable array architectures. In: Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia, pp. 67–72 (2007)
41. Jang, C., Kim, J., Lee, J., Kim, H.S., Yoo, D., Kim, S., Kim, H.S., Ryu, S.: An instruction-scheduling-aware data partitioning technique for coarse-grained reconfigurable architectures. In: Proc. ACM SIGPLAN/SIGBED Conf. Languages, compilers, and tools for embedded systems (LCTES), pp. 151–160 (2011)
42. Karp, R.M., Miller, R.E., Winograd, S.: The organization of computations for uniform recurrence equations. J. ACM **14**(3), 563–590 (1967)
43. Kessler, C.W.: Compiling for VLIW DSPs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)

44. Kim, C., Chung, M., Cho, Y., Konijnenburg, M., Ryu, S., Kim, J.: ULP-SRP: Ultra low power Samsung Reconfigurable Processor for biomedical applications. In: 2012 International Conference on Field-Programmable Technology, pp. 329–334 (2012). DOI 10.1109/FPT.2012.6412157
45. Kim, H.s., Yoo, D.h., Kim, J., Kim, S., Kim, H.s.: An instruction-scheduling-aware data partitioning technique for coarse-grained reconfigurable architectures. In: LCTES '11: Proceedings of the 2011 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems, pp. 151–160 (2011)
46. Kim, W., Choi, Y., Park, H.: Fast modulo scheduler utilizing patternized routes for coarse-grained reconfigurable architectures. ACM Trans. on Architec. and Code Optim. **10**(4), 1–24 (2013)
47. Kim, W., Yoo, D., Park, H., Ahn, M.: SCC based modulo scheduling for coarse-grained reconfigurable processors. In: Proc. Conf. on Field-Programmable Technology, pp. 321–328 (2012)
48. Kim, Y., Kiemb, M., Park, C., Jung, J., Choi, K.: Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization. In: DATE '05: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 12–17 (2005)
49. Kim, Y., Lee, J., Shrivastava, A., Paek, Y.: Operation and data mapping for CGRAs with multi-bank memory. In: LCTES '10: Proceedings of the 2010 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 17–25 (2010)
50. Kim, Y., Lee, J., Shrivastava, A., Yoon, J., Paek, Y.: Memory-aware application mapping on coarse-grained reconfigurable arrays. In: HiPEAC '10: Proceedings of the 2010 International Conference on High Performance Embedded Architectures and Compilers, pp. 171–185 (2010)
51. Kim, Y., Mahapatra, R.: A new array fabric for coarse-grained reconfigurable architecture. In: Proceedings of the IEEE EuroMicro Conference on Digital System Design, pp. 584–591 (2008)
52. Kim, Y., Mahapatra, R., Park, I., Choi, K.: Low power reconfiguration technique for coarse-grained reconfigurable architecture. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **17**(5), 593–603 (2009)
53. Kim, Y., Mahapatra, R.N.: Dynamic Context Compression for Low-Power Coarse-Grained Reconfigurable Architecture. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **18**(1), 15–28 (2010)
54. Lam, M.S.: Software pipelining: an effective scheduling technique for VLIW machines. In: Proc. PLDI, pp. 318–327 (1988)
55. Lambrechts, A., Raghavan, P., Jayapala, M., Catthoor, F., Verkest, D.: Energy-aware interconnect optimization for a coarse grained reconfigurable processor. In: Proceedings of the International Conference on VLSI Design, pp. 201–207 (2008)
56. Lee, G., Choi, K., Dutt, N.: Mapping multi-domain applications onto coarse-grained reconfigurable architectures. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems **30**(5), 637–650 (2011)
57. Lee, J., Seo, S., Lee, H., Sim, H.U.: Flattening-based mapping of imperfect loop nests for CGRAs. In: 2014 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2014, Uttar Pradesh, India, October 12–17, 2014, pp. 9:1–9:10 (2014)
58. Lee, J.e., Choi, K., Dutt, N.D.: An algorithm for mapping loops onto coarse-grained reconfigurable architectures. In: LCTES '03: Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 183–188 (2003)
59. Lee, L.H., Moyer, B., Arends, J.: Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In: ISLPED '99: Proceedings of the 1999 International symposium on Low power electronics and design, pp. 267–269. ACM, New York, NY, USA (1999)

60. Lee, M.H., Singh, H., Lu, G., Bagherzadeh, N., Kurdahi, F.J., Filho, E.M.C., Alves, V.C.: Design and implementation of the MorphoSys reconfigurable computing processor. J. VLSI Signal Process. Syst. **24**(2/3), 147–164 (2000)
61. Lee, W.J., Woo, S.O., Kwon, K.T., Son, S.J., Min, K.J., Jang, G.J., Lee, C.H., Jung, S.Y., Park, C.M., Lee, S.H.: A scalable GPU architecture based on dynamically reconfigurable embedded processor. In: Proc. ACM Conference on High-Performance Graphics (2011)
62. Liang, S., Yin, S., Liu, L., Guo, Y., Wei, S.: A coarse-grained reconfigurable architecture for compute-intensive MapReduce acceleration. Computer Architecture Letters **15**(2), 69–72 (2016)
63. Lin, X., Yin, S., Liu, L., Wei, S.: Exploiting parallelism of imperfect nested loops with sibling inner loops on coarse-grained reconfigurable architectures. In: 21st Asia and South Pacific Design Automation Conference, ASP-DAC 2016, Macao, January 25–28, 2016, pp. 456–461 (2016)
64. Liu, D., Yin, S., Liu, L., Wei, S.: Mapping multi-level loop nests onto CGRAs using polyhedral optimizations. IEICE Transactions **98-A**(7), 1419–1430 (2015)
65. Liu, D., Yin, S., Peng, Y., Liu, L., Wei, S.: Optimizing spatial mapping of nested loop for coarse-grained reconfigurable architectures. IEEE Trans. VLSI Syst. **23**(11), 2581–2594 (2015)
66. Liu, L., Deng, C., Wang, D., Zhu, M., Yin, S., Cao, P., Wei, S.: An energy-efficient coarse-grained dynamically reconfigurable fabric for multiple-standard video decoding applications. In: Proceedings of the IEEE 2013 Custom Integrated Circuits Conference, pp. 1–4 (2013). https://doi.org/10.1109/CICC.2013.6658434
67. Liu, L., Wang, D., Chen, Y., Zhu, M., Yin, S., Wei, S.: An implementation of multiple-standard video decoder on a mixed-grained reconfigurable computing platform. IEICE Transactions **99-D**(5), 1285–1295 (2016)
68. Madhu, K.T., Das, S., Nalesh, S., Nandy, S.K., Narayan, R.: Compiling HPC kernels for the REDEFINE CGRA. In: 17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015, 7th IEEE International Symposium on Cyberspace Safety and Security, CSS 2015, and 12th IEEE International Conference on Embedded Software and Systems, ICESS 2015, New York, NY, USA, August 24–26, 2015, pp. 405–410 (2015)
69. Mahadurkar, M., Merchant, F., Maity, A., Vatwani, K., Munje, I., Gopalan, N., Nandy, S.K., Narayan, R.: Co-exploration of NLA kernels and specification of compute elements in distributed memory CGRAs. In: XIVth International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS 2014, Agios Konstantinos, Samos, Greece, July 14–17, 2014, pp. 225–232 (2014)
70. Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E., Bringmann, R.A.: Effective compiler support for predicated execution using the hyperblock. In: MICRO 25: Proceedings of the 25th annual International symposium on Microarchitecture, pp. 45–54. IEEE Computer Society Press, Los Alamitos, CA, USA (1992)
71. Mei, B., De Sutter, B., Vander Aa, T., Wouters, M., Kanstein, A., Dupont, S.: Implementation of a coarse-grained reconfigurable media processor for AVC decoder. Journal of Signal Processing Systems **51**(3), 225–243 (2008)
72. Mei, B., Lambrechts, A., Verkest, D., Mignolet, J.Y., Lauwereins, R.: Architecture exploration for a reconfigurable architecture template. IEEE Design and Test of Computers **22**(2), 90–101 (2005)
73. Mei, B., Vernalde, S., Verkest, D., Lauwereins, R.: Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: A case study. In: DATE '04: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 1224–1229 (2004)
74. Mei, B., Vernalde, S., Verkest, D., Man, H.D., Lauwereins, R.: ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In: Proc. of Field-Programmable Logic and Applications, pp. 61–70 (2003)

75. Mei, B., Vernalde, S., Verkest, D., Man, H.D., Lauwereins, R.: Exploiting loop-level parallelism for coarse-grained reconfigurable architecture using modulo scheduling. IEE Proceedings: Computer and Digital Techniques **150**(5) (2003)
76. Merchant, F., Maity, A., Mahadurkar, M., Vatwani, K., Munje, I., Krishna, M., Nalesh, S., Gopalan, N., Raha, S., Nandy, S.K., Narayan, R.: Micro-architectural enhancements in distributed memory CGRAs for LU and QR factorizations. In: 28th International Conference on VLSI Design, VLSID 2015, Bangalore, India, January 3–7, 2015, pp. 153–158 (2015)
77. Novo, D., Schuster, T., Bougard, B., Lambrechts, A., Van der Perre, L., Catthoor, F.: Energy-performance exploration of a CGA-based SDR processor. Journal of Signal Processing Systems (2009)
78. Oh, T., Egger, B., Park, H., Mahlke, S.: Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. In: LCTES '09: Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 21–30 (2009)
79. PACT XPP Technologies: XPP-III Processor Overview White Paper (2006)
80. Pager, J., Jeyapaul, R., Shrivastava, A.: A software scheme for multithreading on CGRAs. ACM Trans. Embedded Comput. Syst. **14**(1), 19 (2015)
81. Park, H., Fan, K., Kudlur, M., Mahlke, S.: Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures. In: CASES '06: Proceedings of the 2006 International Conference on Compilers, architecture and synthesis for embedded systems, pp. 136–146 (2006)
82. Park, H., Fan, K., Mahlke, S.A., Oh, T., Kim, H., Kim, H.S.: Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In: PACT '08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 166–176 (2008)
83. Park, H., Park, Y., Mahlke, S.: Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In: MICRO '09: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 370–380 (2009)
84. Park, H., Park, Y., Mahlke, S.A.: A dataflow-centric approach to design low power control paths in CGRAs. In: Proc. IEEE Symp. on Application Specific Processors, pp. 15–20 (2009)
85. Park, J., Park, Y., Mahlke, S.A.: Efficient execution of augmented reality applications on mobile programmable accelerators. In: Proc. Conf. on Field-Programmable Technology, pp. 176–183 (2013)
86. Park, Y., Park, H., Mahlke, S.: CGRA express: accelerating execution using dynamic operation fusion. In: CASES '09: Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp. 271–280 (2009)
87. Park, Y., Park, H., Mahlke, S., Kim, S.: Resource recycling: putting idle resources to work on a composable accelerator. In: CASES '10: Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, pp. 21–30 (2010)
88. Peng, Y., Yin, S., Liu, L., Wei, S.: Battery-aware loop nests mapping for CGRAs. IEICE Transactions **98-D**(2), 230–242 (2015)
89. Peng, Y., Yin, S., Liu, L., Wei, S.: Battery-aware mapping optimization of loop nests for CGRAs. In: The 20th Asia and South Pacific Design Automation Conference, ASP-DAC 2015, Chiba, Japan, January 19–22, 2015, pp. 767–772 (2015)
90. Petkov, N.: Systolic Parallel Processing. North Holland Publishing (1992)
91. P. Raghavan, A. Lambrechts, M. Jayapala, F. Catthoor, D. Verkest, Corporaal, H.: Very wide register: An asymmetric register file organization for low power embedded processors. In: DATE '07: Proceedings of the Conference on Design, Automation and Test in Europe (2007)
92. Rákossy, Z.E., Merchant, F., Aponte, A.A., Nandy, S.K., Chattopadhyay, A.: Efficient and scalable CGRA-based implementation of column-wise Givens rotation. In: IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2014, Zurich, Switzerland, June 18–20, 2014, pp. 188–189 (2014)
93. Rau, B.R.: Iterative modulo scheduling. Tech. rep., Hewlett-Packard Lab: HPL-94-115 (1995)

94. Rau, B.R., Lee, M., Tirumalai, P.P., Schlansker, M.S.: Register allocation for software pipelined loops. In: PLDI '92: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation, pp. 283–299 (1992)
95. Sankaralingam, K., Nagarajan, R., Liu, H., Kim, C., Huh, J., Burger, D., Keckler, S.W., Moore, C.R.: Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. SIGARCH Comput. Archit. News **31**(2), 422–433 (2003)
96. Scarpazza, D.P., Raghavan, P., Novo, D., Catthoor, F., Verkest, D.: Software simultaneous multi-threading, a technique to exploit task-level parallelism to improve instruction- and data-level parallelism. In: PATMOS '06: Proceedings of the 16th International Workshop on Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation, pp. 107–116 (2006)
97. Schlansker, M., Mahlke, S., Johnson, R.: Control CPR: a branch height reduction optimization for EPIC architectures. SIGPLAN Notices **34**(5), 155–168 (1999)
98. Shao, S., Yin, S., Liu, L., Wei, S.: Map-reduce inspired loop parallelization on CGRA. In: IEEE International Symposium on Circuits and Systems, ISCAS 2014, Melbourne, Victoria, Australia, June 1–5, 2014, pp. 1231–1234 (2014)
99. Shen, J., Lipasti, M.: Modern Processor Design: Fundamentals of Superscalar Processors. McGraw-Hill (2005)
100. Shi, R., Yin, S., Liu, L., Liu, Q., Liang, S., Wei, S.: The implementation of texture-based video up-scaling on coarse-grained reconfigurable architecture. IEICE Transactions **98-D**(2), 276–287 (2015)
101. Silicon Hive: HiveCC Databrief (2006)
102. Sudarsanam, A.: Code optimization libraries for retargetable compilation for embedded digital signal processors. Ph.D. thesis, Princeton University (1998)
103. Suh, D., Kwon, K., Kim, S., Ryu, S., Kim, J.: Design space exploration and implementation of a high performance and low area coarse grained reconfigurable processor. In: Proc. on Conf. Field-Programmable Technology, pp. 67–70 (2012)
104. Suzuki, T., Yamada, H., Yamagishi, T., Takeda, D., Horisaki, K., Vander Aa, T., Fujisawa, T., Van der Perre, L., Unekawa, Y.: High-throughput, low-power software-defined radio using reconfigurable processors. IEEE Micro **31**(6), 19–28 (2011)
105. Taylor, M., Kim, J., Miller, J., Wentzla, D., Ghodrat, F., Greenwald, B., Ho, H., Lee, M., Johnson, P., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Frank, V., Amarasinghe, S., Agarwal, A.: The Raw microprocessor: A computational fabric for software circuits and general purpose programs. IEEE Micro **22**(2), 25–35 (2002)
106. Texas Instruments: TMS320C64x Technical Overview (2001)
107. Theocharis, P., De Sutter, B.: A bimodal scheduler for coarse-grained reconfigurable arrays. ACM Trans. on Architecture and Code Optimization **13**(2), 15:1–15:26 (2016)
108. Van Essen, B., Panda, R., Wood, A., Ebeling, C., Hauck, S.: Managing short-lived and long-lived values in coarse-grained reconfigurable arrays. In: FPL '10: Proceedings of the 2010 International Conference on Field Programmable Logic and Applications, pp. 380–387 (2010)
109. Van Essen, B., Panda, R., Wood, A., Ebeling, C., Hauck, S.: Energy-Efficient Specialization of Functional Units in a Coarse-Grained Reconfigurable Array. In: FPGA '11: Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 107–110 (2011)
110. Vander Aa, T., Palkovic, M., Hartmann, M., Raghavan, P., Dejonghe, A., Van der Perre, L.: A multi-threaded coarse-grained array processor for wireless baseband. In: Proc. 9th IEEE Symp. Application Specific Processors, pp. 102–107 (2011)
111. Venkataramani, G., Najjar, W., Kurdahi, F., Bagherzadeh, N., Bohm, W., Hammes, J.: Automatic compilation to a coarse-grained reconfigurable system-on-chip. ACM Trans. Embed. Comput. Syst. **2**(4), 560–589 (2003)
112. van de Waerdt, J.W., Vassiliadis, S., Das, S., Mirolo, S., Yen, C., Zhong, B., Basto, C., van Itegem, J.P., Amirtharaj, D., Kalra, K., Rodriguez, P., van Antwerpen, H.: The TM3270 media-processor. In: MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture, pp. 331–342. IEEE Computer Society, Washington, DC, USA (2005)

113. Woh, M., Lin, Y., Seo, S., Mahlke, S., Mudge, T., Chakrabarti, C., Bruce, R., Kershaw, D., Reid, A., Wilder, M., Flautner, K.: From SODA to scotch: The evolution of a wireless baseband processor. In: MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture, pp. 152–163. IEEE Computer Society, Washington, DC, USA (2008)
114. Programming XPP-III Processors White Paper (2006)
115. Xu, B., Yin, S., Liu, L., Wei, S.: Low-power loop parallelization onto CGRA utilizing variable dual v_{dd}. IEICE Transactions **98-D**(2), 243–251 (2015)
116. Yang, C., Liu, L., Luo, K., Yin, S., Wei, S.: CIACP: A correlation- and iteration- aware cache partitioning mechanism to improve performance of multiple coarse-grained reconfigurable arrays. IEEE Trans. Parallel Distrib. Syst. **28**(1), 29–43 (2017)
117. Yang, C., Liu, L., Wang, Y., Yin, S., Cao, P., Wei, S.: Configuration approaches to improve computing efficiency of coarse-grained reconfigurable multimedia processor. In: 24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2–4 September, 2014, pp. 1–4 (2014)
118. Yang, C., Liu, L., Wang, Y., Yin, S., Cao, P., Wei, S.: Configuration approaches to enhance computing efficiency of coarse-grained reconfigurable array. Journal of Circuits, Systems, and Computers **24**(3) (2015)
119. Yang, C., Liu, L., Yin, S., Wei, S.: Data cache prefetching via context directed pattern matching for coarse-grained reconfigurable arrays. In: Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5–9, 2016, pp. 64:1–64:6 (2016)
120. Yin, S., Gu, J., Liu, D., Liu, L., Wei, S.: Joint modulo scheduling and v_{dd} assignment for loop mapping on dual-v_{dd} CGRAs. IEEE Trans. on CAD of Integrated Circuits and Systems **35**(9), 1475–1488 (2016)
121. Yin, S., Lin, X., Liu, L., Wei, S.: Exploiting parallelism of imperfect nested loops on coarse-grained reconfigurable architectures. IEEE Trans. Parallel Distrib. Syst. **27**(11), 3199–3213 (2016)
122. Yin, S., Liu, D., Liu, L., Wei, S., Guo, Y.: Joint affine transformation and loop pipelining for mapping nested loop on CGRAs. In: Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9–13, 2015, pp. 115–120 (2015)
123. Yin, S., Liu, D., Peng, Y., Liu, L., Wei, S.: Improving nested loop pipelining on coarse-grained reconfigurable architectures. IEEE Trans. VLSI Syst. **24**(2), 507–520 (2016)
124. Yin, S., Yao, X., Liu, D., Liu, L., Wei, S.: Memory-aware loop mapping on coarse-grained reconfigurable architectures. IEEE Trans. VLSI Syst. **24**(5), 1895–1908 (2016)
125. Yin, S., Zhou, P., Liu, L., Wei, S.: Acceleration of nested conditionals on CGRAs via trigger scheme. In: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2015, Austin, TX, USA, November 2–6, 2015, pp. 597–604 (2015)
126. Yin, S., Zhou, P., Liu, L., Wei, S.: Trigger-centric loop mapping on CGRAs. IEEE Trans. VLSI Syst. **24**(5), 1998–2002 (2016)
127. Yoon, J., Ahn, M., Paek, Y., Kim, Y., Choi, K.: Temporal mapping for loop pipelining on a MIMD-style coarse-grained reconfigurable architecture. In: Proceedings of the International SoC Design Conference (2006)
128. Yoon, J.W., Shrivastava, A., Park, S., Ahn, M., Jeyapaul, R., Paek, Y.: SPKM : A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. In: Proc. 13th Asia South Pacific Design Automation Conf. (ASP-DAC), pp. 776–782 (2008)