

# Coarse-Grained Reconfigurable Array Architectures

Bjorn De Sutter, Praveen Raghavan, Andy Lambrechts

**Abstract** Coarse-Grained Reconfigurable Array (CGRA) architectures accelerate the same inner loops that benefit from the high ILP support in VLIW architectures. By executing non-loop code on other cores, however, CGRAs can focus on such loops to execute them 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, meaning that a large number of computations needs to be performed every second. 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 number of consecutive phases  $P = [1, p]$  with different characteristics, performance can be defined in terms of the operating frequencies  $f_p$ , the instructions executed per cycle  $IPC_p$  and the instruction counts  $IC_p$  of each phase, and in terms of the time overhead involved in switching between the phases  $t_{p \rightarrow p+1}$  as follows:

---

Bjorn De Sutter  
Ghent University, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium and Vrije Universiteit Brussel,  
Pleinlaan 2, 1050 Brussel, Belgium, e-mail: bjorn.desutter@elis.ugent.be

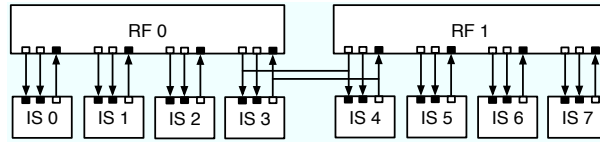
Praveen Raghavan, Andy Lambrechts  
IMEC, Kapeldreef 75, 3001 Heverlee, Belgium e-mail: {ragha,lambrec}@imec.be

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

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  $l \in L \subset P$ , 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, including multimedia and Software-Defined Radio (SDR) applications. Such applications have traditionally employed Very Long Instruction Word (VLIW) architectures such as the TriMedia 3270 [60] and the TI C64 [58], 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, this can be implemented by means of predication [36] instead of with complex control flow. Furthermore, none or very few loops contain multiple exits or continuation points in the form of, e.g., `break` or `continue` statements as in the C-language. 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 [59] or on Ardbeg [62]. 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 (with limited DLP support) are ideally suited for applications of which time-consuming parts have manifest behavior, large amounts of ILP and limited amounts of DLP.

In the remainder of this chapter, Section 2 presents the fundamental properties of CGRAs. Section 3 gives an overview of the design options for CGRAs. This overview helps 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, Section 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



**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.

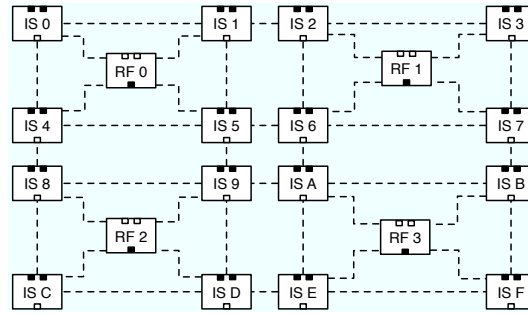
few restrictions on the loop bodies that it can accelerate. Secondly, our use case illustrates how Design Space Exploration (DSE) is necessary to instantiate optimized designs from parameterizable and customizable architecture templates such as the ADRES architecture template. Some conclusions are drawn in Section 5.

## 2 CGRA Basics

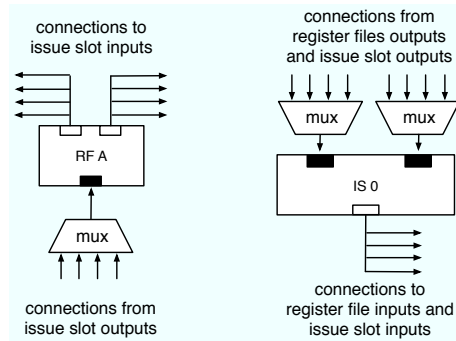
CGRAs focus on the efficient execution of the type of loops discussed in the previous section. By neglecting non-loop code or outer-loop code that is assumed to be executed on other cores, CGRAs can take the VLIW principles for exploiting ILP in loops a step further to consume less energy and deliver 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 [5, 6, 7, 15, 37, 39, 40, 41] CGRA consists of 16 ISs, whereas the TI C64 features 8 slots, and 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 Equation (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 term in Equation (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 into a 2D matrix,



(a) CGRA organization



(b) Connectivity of register files and issue slots

**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).

small RFs with few ports can be distributed in between the ISs as depicted in Figure 2. This contrasts with the many-ported RFs in (clustered) VLIW architectures, which basically feature a one-dimensional design as depicted in Figure 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 [34] 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 then replaces that control flow by data flow by means of predication [36] 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.

There are two main drawbacks to CGRA architectures. Firstly, because they can only execute loops, they need to be coupled to other cores on which all other parts

of the program are executed. In some designs, this coupling introduces run-time and design-time overhead. Secondly, as clearly visible in the example CGRA of Figure 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 Figure 2(b). Consequently more complex compiler technology than that of VLIW compilers 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*

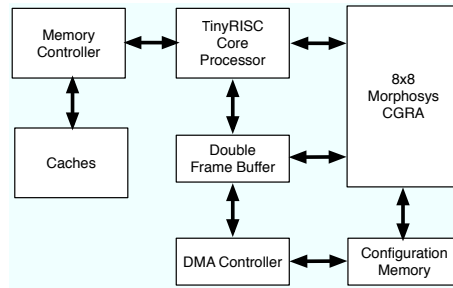
Some CGRA designs are coupled loosely to main processors. For example, Figure 3 depicts how the MorphoSys CGRA [35] is connected as an external accelerator to a TinyRISC Central Processing Unit (CPU). The CPU is responsible for executing non-loop code, for initiating DMA data transfers to and from the 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 [35] depicted in Figure 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.

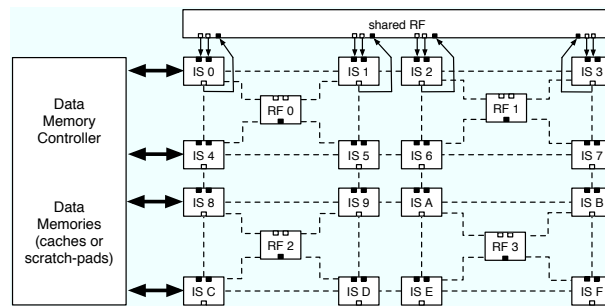
By contrast, an ADRES CGRA is coupled tightly to its main CPU. A simplified ADRES is depicted in Figure 4. Its main CPU is a VLIW consisting of the shared RF and the top row of CGRA ISs. In the main CPU mode, this VLIW executes instructions that are fetched from a VLIW instruction cache and that operate 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. Finally, on the ADRES CGRA, switching between the two modes takes only two cycles. Thus, the run-time overhead is minimal.

Silicon Hive CGRAs [8, 9] do not feature a clear separation between the CGRA accelerator and the main processor. Instead there is just a single processor that can



**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 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.

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 [44], 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

There exist many different mechanisms 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 or dynamic.

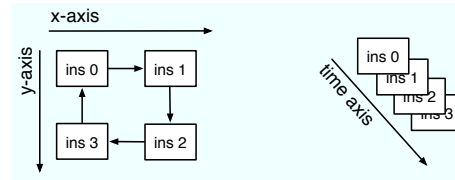
### 3.2.1 Reconfigurability

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 place 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 [24, 25, 26] 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 [14, 18] 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 Figure 5(a). In other words, the mapping problem becomes one of placement and routing, in which instructions and data dependencies between instructions have to be 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 [3].

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 Figure 5(b). Most compiler techniques [15, 17, 20, 39, 43, 45, 46] for these architectures also originate from the FPGA placement and routing world. For CGRAs, the array of resources is not treated as a 2D spatial array, but as





**Fig. 5** Part (a) shows a spatial mapping of a sequence of four instructions on a statically reconfigurable 2x2 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%.

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 [31, 49], with FPGA synthesis-based techniques [3]. Still other compiler techniques exist that are based on constraint solving [55], or on integer-linear programming [1, 64].

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 [34]. 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 [36]. 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 [30]. Furthermore, they also propose to reduce the power consumption in the configuration memories by compressing the configurations [29].

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. Unfortunately, no numbers on the power consumption overhead of this design choice are publicly available.

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 Figure 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 best thus also depends 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).

### 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 [19], or dynamically, similar to the way in which out-of-order processors issue instructions when their operands become available [54]. 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 [23, 51]. 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 [54]. 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 [19]. 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 for VLIW architectures, software pipelining [31, 49] is a very important to expose the ILP in software kernels, so most compiler techniques [15, 17, 20, 39, 43, 45, 46] for statically scheduled CGRAs implement some form of software pipelining.

A final 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 [13] 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.

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 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 [51]. 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 [54] 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 [57] 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.

Other architectures do not support (hardware) multi-threading within one CGRA core at all, like the current ADRES and 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). The advantage is then that each accelerator can be customized for a certain class of loop kernels. Also, ADRES and Silicon Hive are architecture templates, which enables CGRA designers to customize their CGRA cores for the appropriate amount of DLP for each class of loop kernels, in the form of SIMD or subwordparallelism.

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 [52].

### ***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, and with the RFs, other memories and 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 Figure 10 have been published in the past [7, 28, 32, 38]. 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 [47].

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 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 Section 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 [54]. 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 [15, 39] 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 exploit pipelining latches naturally and similarly to using storage space in distributed RFs.

### 3.3.2 Register Files

Compilers for CGRA architectures place operations in 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 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 [7, 38] for DSEs that study both RFs and interconnects.

Besides their size and ports, another important aspect is that RFs can be rotating [50]. The power and delay overhead of rotation is very small in distributed RFs,

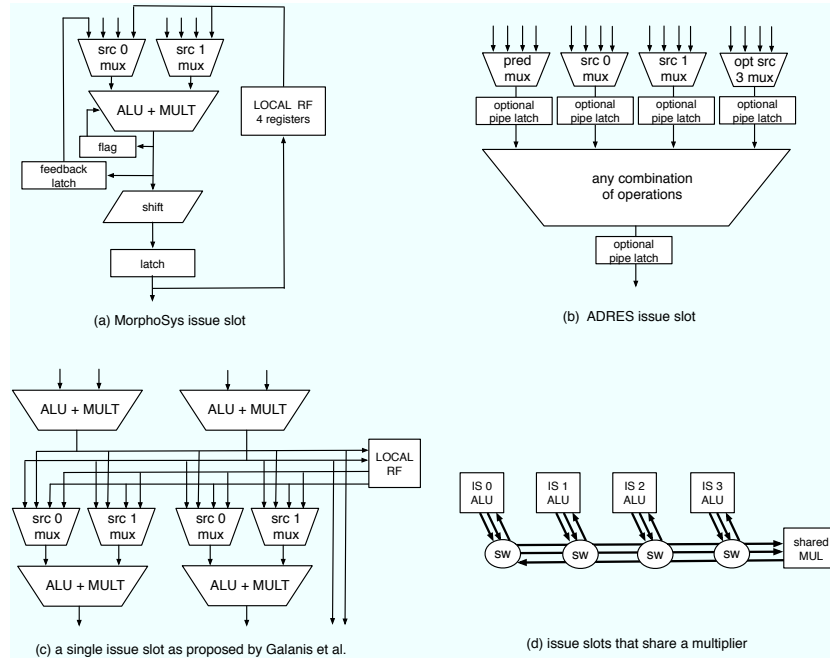
simply because these RFs are small themselves. Still they can provide an important functionality. Consider a dynamically reconfigurable CGRA on which a loop is executed that iterates over  $x$  configurations, i.e., each iteration takes  $x$  cycles. That means that for a write port of an RF, every  $x$  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 Section 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 Figure 2(b) 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 are also used to 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



**Fig. 6** 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. [22], and (d) depicts a row of four ISs that share a multiplier [27].

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.

### 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 6 depicts some of them. For all of the possible designs, it is important to know the context in which these ISs have to operate, such as the interconnects connecting them, the control type of the CGRA, etc.

Figure 6(a) 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 Section 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 Figure 6(b), 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. Given the state of the art in compiler technology for this type of architecture, the advantages of this freedom are (1) the possibility to design different instances optimized for certain application domains, (2) the knowledge that the features of those designs will be exploited, and (3) the ability to compile loops that feature other forms of ILP than DLP. DLP can still be supported, of course, by simply incorporating SIMD-capable (a.k.a. subwordparallel) ISs of, e.g., 4x16 bits wide. The drawback is this design freedom is that, at least with the current compiler techniques, these techniques are so generic that they miss optimization opportunities because they do not exploit regularity in the designed architectures. They do not exploit it for speeding up the compilation, nor do they for producing better schedules.

Figure 6(c) depicts the IS proposed by Galanis et al. [22]. 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 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.

With respect to utilization, it is clear that the designs of Figure 6(a) and 6(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 [27]. Figure 6(d) 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 3 pure ALU ISs and 1 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 subwordparallelism. As illustrated in section 4.2.2, a typical ADRES would consist of 4x4 ISs [6, 37]. TRIPS also features 4x4 ISs. MorphoSys provides 8x8 ISs, but that is because the DLP is implemented as SIMD over multiple ISs, rather than as subwordparallelism within ISs. In our experience, scaling dynamically reconfigurable CGRA architectures such as ADRES to very large arrays (8x8 or larger) does not make sense even with scalable interconnects like mesh or mesh-plus interconnects. Even in loops with high ILP, utilization drops significantly on such large arrays [42]. It is not yet clear what is causing this lower utilization, and there might be several reasons. These include a lack of memory bandwidth, the possibility that the compiler techniques [15, 39] 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 4x4 array, i.e., to 25% of all ISs, while it is connected to 4 out of 64 ISs on an 8x8 array, i.e., to 6.25% of all ISs.

To finalize this section, we want to mention that, just like in any other type of processor, it makes sense to pipeline complex combinatorial logic, e.g., as found in multipliers. There are no fundamental problems to do this, and it can lead to significant increases in utilization and clock frequency.

### 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. Exploiting those automatically in a compiler has not yet been fully solved.

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 [2]. 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 [35]. 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 [48]. However,

this requires the programmer or compiler to perform the data layout in memory in order to exploit the large bandwidth between the level-1 memory and the datapath.

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 [6]. 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, that accesses the same memories in code not software-pipelined, the conflict resolution is disabled to obtain shorter access latencies.

### 3.6 *Compiler Support*

Apart from the specific algorithms used to compile code, the major distinctions between the different existing compiler techniques 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. Because most compiler research has been done to generate static schedules for CGRAs, we focus on those in this section. As already indicated in Sections 3.2.1 and 3.2.2, many algorithms are based on FPGA placement and routing techniques [3] in combination with VLIW code generation techniques like modulo scheduling [31, 49] and hyperblock formation [36].

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) and effectiveness (quality of generated code) 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 in the schedule, placement refers to finding the right IS in that cycle, and routing refers to finding connections to transfer data from producing operations to consuming operations. 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 [49] 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 [45] models whole ISs or whole RFs with single vertices, whereas the simulated-annealing technique in the DRESC [15, 39] compiler that 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. Some techniques, such as DRESC, are built on the central idea of finding the best routes to steer the placement and scheduling, thus exploring many possible routings, while others [20, 43, 45, 46] use heuristics to place and schedule the code, using routability as a constraint during the scheduling. The latter are typically much more efficient, but less effective.

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. To support different programming languages like C and Fortran, the techniques only require a compiler front-end that is able to generate DDGs for the loop bodies. Obviously, the appropriate loop transformations need to be applied before generating the DDG in order to generate one that maps well onto the MRRG of the architecture. Such loop transformations are discussed in detail in Section 4.1.

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 [10, 17, 63] 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 [61]. 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 RSPA architecture, both for spatial [1] and for temporal mapping [64]. 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 [55], of which no detailed information is public.

Code generation techniques for CGRAs based on instruction-selection pattern matching and list-scheduling techniques have also been proposed [21, 22]. It is unclear to what extent these techniques rely on 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 Figure 6(c). Similarly, the static reconfiguration code generation technique by Lee et al. relies on CGRA rows consisting of identical ISs [33]. 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 [12].

One rule-of-thumb covers all the mentioned techniques: more generic techniques, i.e., techniques that are more flexible in targeting different architectures or different instances of an architecture template, are less efficient and often less effective in exploiting special architecture features. In other words techniques that rely on specific hardware features, such as interconnect regularities or specific forms of ISs clustering, while being less flexible, will generally be able to target those hardware features more efficiently, and often also more effectively. Vice versa, architectures with such features usually need specialized compiler techniques. This is similar to the situation of more traditional DSP or VLIW architectures.

## 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 [5, 6, 7, 15, 37, 39, 40, 41] 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 Section 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 [56]. 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 onto 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 [31, 49]. 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

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

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 Equation (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 9 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 Equation (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 Equation (2) to obtain higher IPCs in Equation (1), it is also important to consider the terms  $t_{p \rightarrow p+1}$  in Equation (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 Equation (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 opti-

mized 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 [4] 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 [33], 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 Section 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

Modulo scheduling techniques for CGRAs [15, 17, 20, 39, 45, 46] 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 [36]. 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.

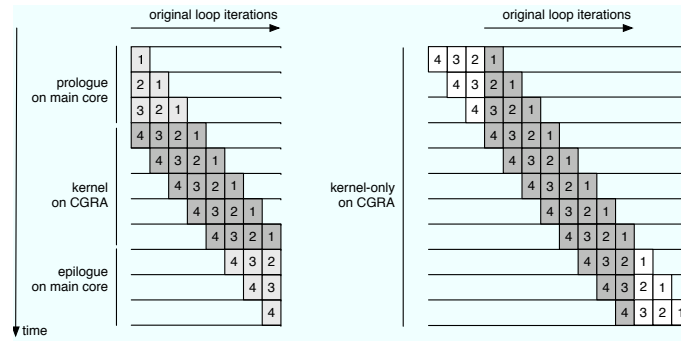
One caveat with this loop splitting is that it causes code size expansion in the CGRA instruction memories. For power consumption reasons, these memories are kept as small as possible. This means that the local improvements obtained with the loop splitting need to be balanced with the total code size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

### 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 Figure 7. 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,





**Fig. 7** 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  $H$  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.

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 Figure 7. 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 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 [53] 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 2.

### 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

**(a) original 15-tap FIR filter**

```

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) filter after loop unrolling, with hard-coded constants**

```

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) after redundant memory accesses are eliminated**

```

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. 8** Three C versions of a FIR filter.

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. The three C implementations of a Finite Impulse Response (FIR) filter in Figure 8 provide an excellent example.

Figure 8(a) depicts a FIR implementation that is efficient for architectures with few registers. For architectures with more registers, the implementation depicted in Figure 8(b) will usually be more efficient, as many memory accesses have been eliminated. Finally, the equivalent code in Figure 8(c) 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 1 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

program	cycle count		memory accesses	
	CGRA	TI C64+	CGRA	TI C64+
FIR (a)	11828	1054	6221	1618
FIR (b)	1247	1638	3203	2799
FIR (c)	664	10062	422	416

**Table 1** 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.

the code of Figure 8(c) 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.

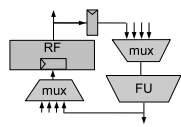
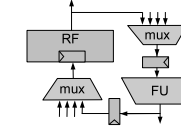
## 4.2 ADRES Design Space Exploration

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 Section 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 [37] and one for SDR baseband processing [5, 6]. Their main differences are presented in Table 2. 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

	<b>multimedia CGRA</b>	<b>SDR CGRA</b>
# 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		
power, clock, and area	91 mW at 300 MHz for 4mm <sup>2</sup>	310mW at 400 MHz for 5.79mm <sup>2</sup>

**Table 2** Main differences between two studied ADRES CGRAs. 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.

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 2. 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 2 hence directly contribute in the maximization of the factor  $f_p$  in Equation (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 Equation (1), this can result in lower performance. In CGRAs, however, this is not necessarily the case, as explained in Section 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

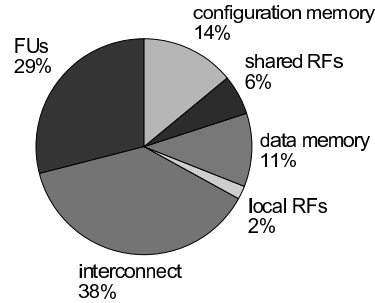
Benchmark	CGRA	Loop	#ops	ResMII	pipelined			non-pipelined		
					RecMII	II	IPC	RecMII	II	IPC
AVC decoder	multimedia	MBFilter1	70	5	2	6	11.7	1	6	11.7
		MBFilter2	89	6	7	9	9.9	6	8	11.1
		MBFilter3	40	3	3	4	10.0	2	3	13.3
		MBFilter4	105	7	2	9	11.7	1	9	11.7
		MotionComp	109	7	3	10	10.9	2	10	10.9
		FindFrameEnd	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
		<b>Average</b>								
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
		<b>Average</b>					<b>12.1</b>			
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
<b>Average</b>					<b>11.7</b>				<b>11.6</b>	
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
		<b>Average</b>					<b>11.6</b>			
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
		<b>Average</b>					<b>11.1</b>			

**Table 3** Results for the benchmark loops. 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.

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 90nm 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



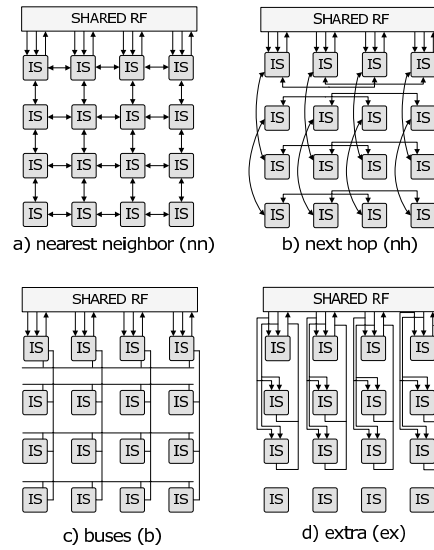
**Fig. 9** Average power consumption distribution of the ADRES SDR CGRA in CGRA mode.

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 2. More detailed results of these experiments are available in the literature [5, 6] for this SDR ADRES instance, as well as for the multimedia instance [37]. The SDR ADRES instance has also been produced in silicon in samples of a full SoC SDR chip [16]. 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 Figure 9, 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 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, 11, 32, 37, 42]. We present one experimental result [32] 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 in-

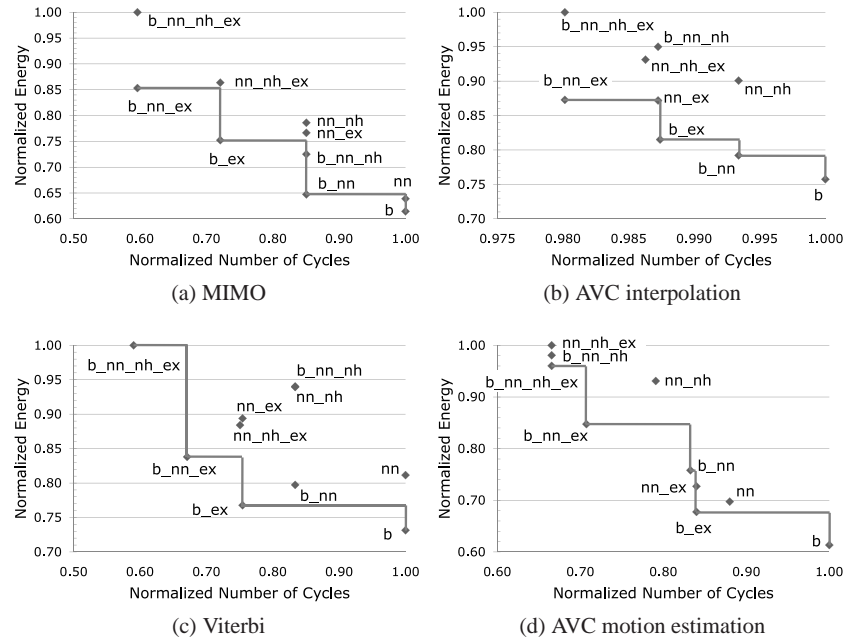


**Fig. 10** Basic interconnects that can be combined. All bidirectional edges between two ISs denote that all outputs of one IS are connected to the inputs of the other IS and vice versa. Buses that connect all connected IS outputs to all connected IS inputs are shown as edges without arrows.

ner 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 Figure 10, 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 Figure 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 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.



**Fig. 11** DSE results for four microbenchmarks on 4x4 CGRAs with fixed ISs and fixed RFs, but with varying interconnects.

## 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.



## References

1. AHN, M., YOON, J. W., PAEK, Y., KIM, Y., KIEMB, M., AND 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* (3001 Leuven, Belgium, Belgium, 2006), European Design and Automation Association, pp. 363–368.
2. BARUA, R. *Maps: a compiler-managed memory system for software-exposed architectures*. PhD thesis, Massachusetts Institute of Technology, January 2000.
3. BETZ, V., ROSE, J., AND MARGUARDT, A. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.
4. BONDALAPATI, K. Parallelizing DSP nested loops on reconfigurable architectures using data context switching. In *DAC '01: Proceedings of the 38th annual Design Automation Conference* (New York, NY, USA, 2001), ACM, pp. 273–276.
5. BOUGARD, B., DE SUTTER, B., RABOU, S., NOVO, D., ALLAM, O., DUPONT, S., AND 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* (New York, NY, USA, 2008), ACM, pp. 716–721.
6. BOUGARD, B., DE SUTTER, B., VERKEST, D., VAN DER PERRE, L., AND LAUWEREINS, R. A coarse-grained array accelerator for software-defined radio baseband processing. *IEEE Micro* 28, 4 (2008), 41–50.
7. BOUWENS, F., BEREKOVIC, M., GAYDADJIEV, G., AND DE SUTTER, B. Architecture enhancements for the ADRES coarse-grained reconfigurable array. In *Proc. of HiPEAC Conf.* (2008).
8. BURNS, G., AND GRUIJTERS, P. Flexibility tradeoffs in SoC design for low-cost SDR. *Proceedings of SDR Forum Technical Conference* (2003).
9. BURNS, G., GRUIJTERS, P., HUISKENS, J., AND VAN WEL, A. Reconfigurable accelerators enabling efficient SDR for low cost consumer devices. *Proceedings of SDR Forum Technical Conference* (2003).
10. CARDOSO, J. M. P., AND WEINHARDT, M. XPP-VC: A C compiler with temporal partitioning for the PACT-XPP architecture. In *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications* (London, UK, 2002), Springer-Verlag, pp. 864–874.
11. CERVERO, T., KANSTEIN, A., LÓPEZ, S., DE SUTTER, B., SARMIENTO, R., AND MIGNOLET, J.-Y. Architectural exploration of the H.264/AVC decoder onto a coarse-grain reconfigurable architecture. In *Proc. of the International Conference on Design of Circuits and Integrated Systems* (2008).
12. COONS, K. E., CHEN, X., BURGER, D., MCKINLEY, K. S., AND KUSHWAHA, S. K. A spatial path scheduling algorithm for edge architectures. *SIGPLAN Not.* 41, 11 (2006), 129–140.
13. CORPORAAL, H. *Microprocessor Architectures from VLIW to TTA*. John Wiley, 1998.
14. CRONQUIST, D., FRANKLIN, P., FISHER, C., FIGUEROA, M., AND EBELING, C. Architecture design of reconfigurable pipelined datapaths. In *Proceedings of the Twentieth Anniversary Conference on Advanced Research in VLSI* (1999).
15. DE SUTTER, B., COENE, P., VANDER AA, T., AND 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* (New York, NY, USA, 2008), ACM, pp. 151–160.
16. 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., AND DER PERRE, L. V. A 200Mbps+ 2.14nJ/b digital baseband multi processor system-on-chip for SDRs. In *Proc of VLSI Symposium* (June 2009).
17. EBELING, C. Compiling for coarse-grained adaptable architectures. Tech. Rep. UW-CSE-02-06-01, University of Washington, 2002.

18. EBELING, C. The general RaPiD architecture description. Tech. Rep. UW-CSE-02-06-02, University of Washington, 2002.
19. FISHER, J., FARABOSCHI, P., AND YOUNG, C. *Embedded Computing, A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2005.
20. FRIEDMAN, S., CARROLL, A., VAN ESSEN, B., YLVISAKER, B., EBELING, C., AND HAUCK, S. SPR: an architecture-adaptive CGRA mapping tool. In *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays* (New York, NY, USA, 2009), ACM, pp. 191–200.
21. GALANIS, M. D., MILIDONIS, A., THEODORIDIS, G., SOUDRIS, D., AND GOUTIS, C. E. A method for partitioning applications in hybrid reconfigurable architectures. *Design Automation for Embedded Systems 10*, 1 (2006), 27–47.
22. GALANIS, M. D., THEODORIDIS, G., TRAGOUDAS, S., AND GOUTIS, C. E. A reconfigurable coarse-grain data-path for accelerating computational intensive kernels. *Journal of Circuits, Systems and Computers (JCSC)* (2005), 877–893.
23. 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., AND 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* (New York, NY, USA, 2009), ACM, pp. 1–12.
24. HARTENSTEIN, R., HERZ, M., HOFFMANN, T., AND NAGELDINGER, U. Mapping applications onto reconfigurable kressarrays. In *Proceedings of the 9th International Workshop on Field Programmable Logic and Applications* (1999).
25. HARTENSTEIN, R., HERZ, M., HOFFMANN, T., AND NAGELDINGER, U. Generation of design suggestions for coarse-grain reconfigurable architectures. In *Proceedings of the 10th International Workshop on Field Programmable Logic and Applications* (2000).
26. HARTENSTEIN, R., HOFFMANN, T., AND 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).
27. KIM, Y., KIEMB, M., PARK, C., JUNG, J., AND 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* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 12–17.
28. KIM, Y., AND MAHAPATRA, R. A new array fabric for coarse-grained reconfigurable architecture. In *Proceedings of the IEEE EuroMicro Conference on Digital System Design* (2008), pp. 584–591.
29. KIM, Y., AND MAHAPATRA, R. Dynamic context compression for low-power coarse-grained reconfigurable architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2009). To appear.
30. KIM, Y., MAHAPATRA, R., PARK, I., AND CHOI, K. Low power reconfiguration technique for coarse-grained reconfigurable architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17, 5 (2009), 593–603.
31. LAM, M. S. Software pipelining: an effective scheduling technique for VLIW machines. In *Proc. PLDI* (1988), pp. 318–327.
32. LAMBRECHTS, A., RAGHAVAN, P., JAYAPALA, M., CATHOOR, F., AND VERKEST, D. Energy-aware interconnect optimization for a coarse grained reconfigurable processor. *VLSI Design, International Conference on 0* (2008), 201–207.
33. LEE, J.-E., CHOI, K., AND DUTT, N. D. An algorithm for mapping loops onto coarse-grained reconfigurable architectures. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems* (New York, NY, USA, 2003), ACM, pp. 183–188.
34. LEE, L. H., MOYER, B., AND 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* (New York, NY, USA, 1999), ACM, pp. 267–269.

35. LEE, M.-H., SINGH, H., LU, G., BAGHERZADEH, N., KURDAHI, F. J., FILHO, E. M. C., AND ALVES, V. C. Design and implementation of the MorphoSys reconfigurable computing processor. *J. VLSI Signal Process. Syst.* 24, 2/3 (2000), 147–164.
36. MAHLKE, S. A., LIN, D. C., CHEN, W. Y., HANK, R. E., AND BRINGMANN, R. A. Effective compiler support for predicated execution using the hyperblock. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture* (Los Alamitos, CA, USA, 1992), IEEE Computer Society Press, pp. 45–54.
37. MEI, B., DE SUTTER, B., VANDER AA, T., WOUTERS, M., KANSTEIN, A., AND DUPONT, S. Implementation of a coarse-grained reconfigurable media processor for AVC decoder. *Journal of Signal Processing Systems* 51, 3 (2008), 225–243.
38. MEI, B., LAMBRECHTS, A., VERKEST, D., MIGNOLET, J.-Y., AND LAUWEREINS, R. Architecture exploration for a reconfigurable architecture template. *IEEE Design and Test of Computers* 22, 2 (2005), 90–101.
39. MEI, B., VERNALDE, S., VERKEST, D., AND LAUWEREINS, R. Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: A case study. In *Proc. of Design, Automation and Test in Europe (DATE)* (2004), pp. 1224–1229.
40. MEI, B., VERNALDE, S., VERKEST, D., MAN, H. D., AND LAUWEREINS, R. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Proc. of Field-Programmable Logic and Applications* (2003), pp. 61–70.
41. MEI, B., VERNALDE, S., VERKEST, D., MAN, H. D., AND LAUWEREINS, R. Exploiting loop-level parallelism for coarse-grained reconfigurable architecture using modulo scheduling. *IEE Proceedings: Computer and Digital Techniques* 150, 5 (2003).
42. NOVO, D., SCHUSTER, T., BOUGARD, B., LAMBRECHTS, A., VAN DER PERRE, L., AND CATTLOOR, F. Energy-performance exploration of a CGA-based SDR processor. *Journal of Signal Processing Systems* (2008).
43. OH, T., EGGER, B., PARK, H., AND 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* (New York, NY, USA, 2009), ACM, pp. 21–30.
44. PACT XPP TECHNOLOGIES. *XPP-III Processor Overview White Paper*, 2006.
45. PARK, H., FAN, K., KUDLUR, M., AND 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* (New York, NY, USA, 2006), ACM, pp. 136–146.
46. PARK, H., FAN, K., MAHLKE, S. A., OH, T., KIM, H., AND 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* (New York, NY, USA, 2008), ACM, pp. 166–176.
47. PETKOV, N. *Systolic Parallel Processing*. North Holland Publishing, 1992.
48. P.RAGHAVAN, A.LAMBRECHTS, M.JAYAPALA, F.CATTLOOR, D.VERKEST, AND 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).
49. RAU, B. R. Iterative modulo scheduling. Tech. rep., Hewlett-Packard Lab: HPL-94-115, 1995.
50. RAU, B. R., LEE, M., TIRUMALAI, P. P., AND 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* (1992), pp. 283–299.
51. SANKARALINGAM, K., NAGARAJAN, R., LIU, H., KIM, C., HUH, J., BURGER, D., KECKLER, S. W., AND MOORE, C. R. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. *SIGARCH Comput. Archit. News* 31, 2 (2003), 422–433.
52. SCARPAZZA, D. P., RAGHAVAN, P., NOVO, D., CATTLOOR, F., AND VERKEST, D. Software simultaneous multi-threading, a technique to exploit task-level parallelism to improve instruction- and data-level parallelism. In *Proceedings of the 16th International Workshop on*

- Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation (PATMOS)* (2006), pp. 107–116.
53. SCHLANSKER, M., MAHLKE, S., AND JOHNSON, R. Control CPR: a branch height reduction optimization for EPIC architectures. *SIGPLAN Not.* 34, 5 (1999), 155–168.
  54. SHEN, J., AND LIPASTI, M. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill, 2005.
  55. SILICON HIVE. *HiveCC Databrief*, 2006.
  56. SUDARSANAM, A. *Code optimization libraries for retargetable compilation for embedded digital signal processors*. PhD thesis, Princeton University, 1998.
  57. 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., AND AGARWAL, A. The Raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro* 22, 2 (2002), 25–35.
  58. TEXAS INSTRUMENTS. *TMS320C64x Technical Overview*, 2001.
  59. VAN BERKEL, K., HEINLE, F., AMD MEUWISSEN, P., MOERMAN, K., AND WEISS, M. Vector processing as an enabler for software-defined radio in handheld devices. *EURASIP Journal on Applied Signal Processing* 2005, 16 (2005), 2613–2625.
  60. 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., AND VAN ANTWERPEN, H. The TM3270 media-processor. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 331–342.
  61. VENKATARAMANI, G., NAJJAR, W., KURDAHI, F., BAGHERZADEH, N., BOHM, W., AND HAMMES, J. Automatic compilation to a coarse-grained reconfigurable system-on-chip. *ACM Trans. Embed. Comput. Syst.* 2, 4 (2003), 560–589.
  62. WOH, M., LIN, Y., SEO, S., MAHLKE, S., MUDGE, T., CHAKRABARTI, C., BRUCE, R., KERSHAW, D., REID, A., WILDER, M., AND 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* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 152–163.
  63. *Programming XPP-III Processors White Paper*, 2006.
  64. YOON, J., AHN, M., PAEK, Y., KIM, Y., AND K., C. Temporal mapping for loop pipelining on a MIMD-style coarse-grained reconfigurable architecture. In *Proc. International SoC Design Conference* (2006).

# Index

- accelerator, 6
- architecture for dynamically reconfigurable embedded systems (ADRES), 20
- coarse-grained reconfigurable array (CGRA), 2
- compiler optimization, 21
- data dependence graph (DDG), 19
- data-level parallelism (DLP), 11
- design space exploration (DSE), 27
- finite impulse response (FIR), 26
- instruction-level parallelism (ILP), 2
- interconnects, 12, 30
- kernel-only loops, 24
- loop transformations, 22
  - loop combination, 23
  - loop fusion, 23
  - loop interchange, 23
  - loop unrolling, 22
- modulo routing resource graph (MRRG), 18
- modulo scheduling, 21
- multimedia processing, 27
- Pareto, 31
- placement and routing, 18
- predication, 24
- reconfigurable, 8
- routing, 18
- routing resource graph (RRG), 18
- scheduling, 18
- single instruction multiple data (SIMD), 9, 12
- software pipelining, 21
- software-defined radio (SDR), 27
- very long instruction word (VLIW), 4