2011 International Conference on Parallel Processing

Enabling Multithreading on CGRAs

†Aviral Shrivastava, †Jared Pager, †Reiley Jeyapaul, †♯Mahdi Hamzeh, and ♯Sarma Vrudhula
†Compiler Microarchiteture Lab, ♯VLSI Electronic Design Automation Laboratory
Arizona State University, Tempe, AZ, USA
Email: {aviral.shrivastava, jppager, reiley.jeyapaul, mahdi.hamzeh, vrudhula}@asu.edu

***Abstract*—Coarse-Grained Reconfigurable Arrays or CGRAs are programmable fabrics that promise both high performance and high power efficiency. Traditionally, CGRAs were used to accelerate extremely-embedded systems, and were typically manually programmed. However, as CGRAs are conceived to be used as more general-purpose accelerators, there is a need to develop software tools and capabilities. Much work has been done on developing compiler techniques for CGRAs, making programming them easier; however, there is no support for multithreading. As an accelerator to a multithreaded processor, CGRAs now are restricted to accelerating only one kernel of one thread running on the processor at any point in time. Supporting multithreading is difficult, since the start times and end times of threads are dynamic in nature, while CGRAs are statically scheduled. In this paper, we propose a strategy to do multithreading on a CGRA. The chief capability that we develop is a scheme to quickly transform an existing application mapping using the entire CGRA to one using only a fraction of it. Our experimental results on kernels from multimedia applications demonstrate that multithreading support can improve the total throughput of a CGRA by over 30%, 75%, and 150% on 4x4, 6x6, and 8x8 CGRAs, respectively, compared to single-threaded methods.**

I. Introduction

Power efficiency has become one of the most important design metrics in many computational domains. In high performance computing, performance is critically constrained by power and thermal factors such that greater performance is only achievable by increasing power efficiency. In addition, power efficiency is arguably the most important metric in determining the usability of consumer electronic devices, such as cell phones, music players, tablets, etc. Here, power efficiency directly translates into system weight and volume (since battery weight and volume is the majority constituent of system weight and volume), recharge time, and processing frequency of the device.

Coarse-Grained Reconfigurable Arrays or CGRAs are a promising solution for power efficient computation. A CGRA is a grid of very efficient processors, typically nothing more than an Arithmetic Logic Unit (ALU) and a small register file (RF). Computation is statically mapped out on the CGRA during compilation. Very little power is expended in performing an operation and therefore CGRAs are very power efficient. CGRAs have been shown to achieve power efficiencies of 10-100 GOps/W [1]. This is about 2 orders of magnitude higher than the Intel Core i7 (quad core) processor, which has a peak performance of 45 GOps/s, but consumes 130 W of power, providing a power efficiency of 0.347 GOps/W [2]. Several implementations of CGRAs such as MorphoSys [1], ADRES [3], RSPA [4], and KressArray [5] exist. [6] contains a comprehensive summary of many of them.

Initially, CGRAs were used for fast and power efficient processing of streaming applications in multimedia, signal processing, and networking domains. These extremely-embedded systems had a small set of applications, with deterministic computation needs, allowing CGRAs to be programmed by hand. However, as the need for power efficiency grows in all computing domains, researchers have started to conceive the use of CGRAs as more general-purpose accelerators. Here, the CGRA would be a tightly-coupled accelerator to a processor with the ability to accelerate exponentially more application kernels than present in extremely-embedded systems. In order to automate this process, a lot of research in developing automated compiler techniques to map a given loop kernel onto a CGRA, e.g., [7], [8] has been undertaken since the turn of this century.

As an accelerator to a processor, a CGRA can only accelerate one kernel of one thread running on the processor at any given point in time. This is because CGRAs are completely statically scheduled, while thread start and end times are extremely dynamic in nature. CGRA compilers typically map the loop kernel to the entire CGRA, preventing any other thread from using the CGRA. A support for multithreading in CGRAs will not only increase CGRA resource utilization, and therefore throughput, but also improve the performance of communicating tasks, and help alleviate memory bottlenecks.

A key requirement for multithreading is the ability to restrict a given kernel to use only a portion of the CGRA. However, at compile time, the compiler will compile using the entire CGRA. This requires the ability to shrink an existing schedule to use less of the CGRA at runtime. The multithreading mechanism on the processor can then shrink and expand the schedules dynamically as threads are invoked and finish.

One challenge in this is that the schedule transformation problem is equivalent to the original kernel mapping problem. However, this is difficult and the compilation time of existing CGRA compilers is quite long, using techniques like simulated annealing [9]. Traditionally, compile time has not been a concern, as the applications are compiled only once and ran indefinitely. However, to support multithreading, the schedule transformation algorithm must be fast, since it will be used at runtime.

In this paper, we propose an application mapping and dynamic transformation scheme that enables multithreading capabilities on a CGRA structure. The key idea in this

0190-3918/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPP.2011.77

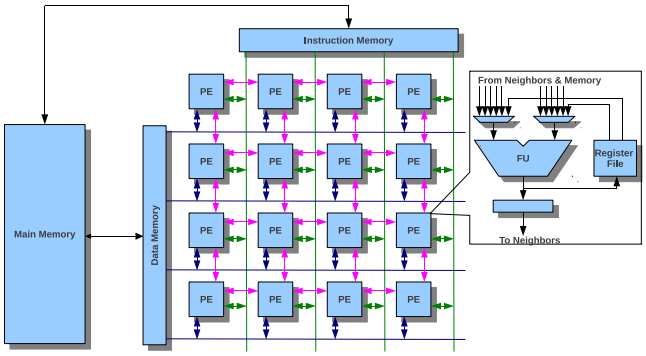


Fig. 1. A 4x4 CGRA is essentially a 2D grid of PEs, and data and instruction memories. Each PE can operate on the results of its neighboring PEs.A PE is essentially an ALU with a local register file.There is a global storage area reserved by the compiler in the Data Memory.

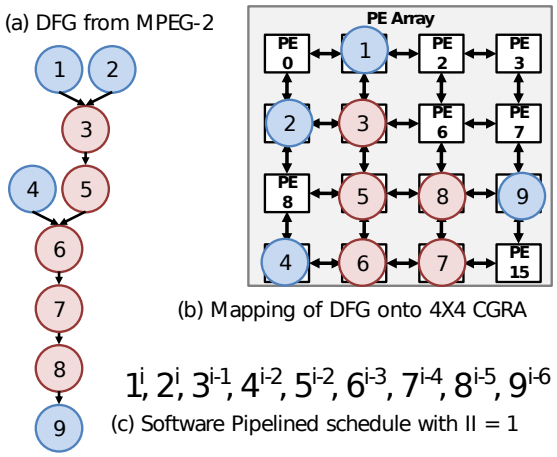


Fig. 2. (a) The loop kernel of application can be represented as a data flow graph (DFG). The vertices of DFG are mapped to PEs, and edges to interconnects between PEs. (c) Software pipelined schedule with II = 1 of the DFG. The schedule shows that operations 1 and 2 of i^{th} iteration can be executed along-with operation 3 of iteration $i-1$, and so on for other operations. Operation 3 is executing on the output of operations 1 and 2 from the previous cycle. (b) The mapping of the DFG on 4X4 CGRA.

paper is to add some additional constraints to the compiler when it is generating the original schedule (for the whole CGRA), such that i) additional mapping constraints allows for optimal and quick transformation, and ii) additional mapping constraints maintain the original mapping quality well. Our experimental results indicate that our added constraints are reasonable and minimally affects the quality of the original mapping. Given this constraint on the original mapping, we develop a transformation that will shrink the original schedule to i) use less of the CGRA in low-order polynomial time and ii) maintain optimality of mapping, i.e., using $frac$ of the original CGRA causes an increase in execution time of only $\frac{1}{frac}$ for that schedule. Our experiments demonstrate the effectiveness of multithreading on the CGRA in media-based benchmark applications, showing performance improvements of over 30%, 75%, and 150% on 4x4, 6x6, and 8x8 CGRAs, respectively, compared to single-threaded approaches.

II. CGRA Architecture and Kernel Mapping

A CGRA is essentially an array of processing elements (PEs), connected by a 2-D mesh-like network. As illustrated in Figure 1, each PE can execute an arithmetic or logic operation such as addition, shift, multiplication, or load/store every cycle. PEs can load (store) data from (to) the on-chip local memory, but they can also operate on the output of a neighboring PE in the next cycle through the interconnect network. Each PE has a local rotating register file (RF) structure to store constants and temporary values used by the PE for operations. The rotation register file is essential for modulo scheduling [10], which is commonly used for application mapping. The internal structure of a single PE is also shown in Figure 1, where one of many data path connections is shown. The data stored in the register file structure of one PE can also be made available to a neighboring PE in the next cycle, thereby facilitating routing of data within the CGRA PEs. The functionality of a PE, i.e. the choice of source operands, destination of the result, and the arithmetic/logic operation is specified in the configuration, which is generated by the compiler during application mapping.

CGRAs are typically used to accelerate the innermost loops of applications in a power efficient manner. The innermost loop of a perfectly nested loop can be represented as a data flow graph (DFG), in which the set of vertices are micro-operations, and the edges denote the data dependencies between the operations. A loop kernel from MPEG2 is shown in Figure 2, in which nodes 1, 2, and 4 are load operations, node 9 a store, and the rest arithmetic or logic operations. While not the case in this loop, the data dependencies can also be loop-carried. To map a DFG, the loop must be first software pipelined. Most of the existing mapping techniques use modulo scheduling. Modulo scheduling [11] is a software pipelining technique that overlaps iterations of a loop to exploit parallelism. The schedule repeats in a modulo fashion. The common goal is to minimize the initiation interval (II). II is defined as the interval between two successive iterations of a loop. In Figure 2, the software pipelined schedule shows that when the operation 1 of iteration i executes, operation 2 of iteration i, operation 3 of iteration $i-1$ etc. must be executed.

The task of mapping a software pipelined application onto a CGRA comprises of mapping the operations of the data flow graph (DFG) onto the PEs of the CGRA, and mapping the edges of the data dependency graph onto interconnect paths on the CGRA. A path is a list of adjacent connections in the CGRA (adjacent connections share a PE), with the length of the path being defined as the number of connections. Ideally we want to map an edge in the DFG to a path of length one, i.e., one connection between PEs. For example, the edge from operation 2 to 3 is mapped on the interconnection between PEs 4 and 5. However, due to limited inter-connectivity, this is not always possible. Larger paths include PEs between the adjacent connections. These PEs, termed routing PEs can only transfer input data to its outputs. In addition, they lengthen

the schedule. Consequently, the software pipelining of the loop must be done together with the scheduling, mapping and routing of the DFG, which is an NP-complete problem. This makes application mapping challenging and compute-intensive.

III. Related Work

The power and performance benefits of CGRAs have attracted several researchers, and many CGRA architectures have been proposed over time. Most of the research on CGRAs over the past decade was concerned with designing CGRAs, including PipeRench [12], PADDI [13], XPP [14], REMARC [15], MATRIX [16], KressArray [5] and Morphosys [1]. A very good survey of CGRA architectures is presented in [6].

CGRAs are similar to systolic arrays [17], having a similar structure consisting of a set of functional units connected in a regular network. However, systolic arrays have traditionally been used as application specific hardware accelerators, while CGRAs are more general-purpose accelerators. Systolic arrays have limited programming capability, lacking a generic instruction and data memory. The typical model of computation of systolic arrays is streaming, i.e., data comes from one side and operations are applied to data. Access to the memory is more restricted in systolic arrays where only the closest PEs have access to the memory. In contrast, there is an explicit instruction and data memory, and a shared data bus for each row of the CGRA, giving many more PEs access to the memory. Systolic arrays and CGRAs can be contrasted as follows: Design the systolic array to the application [17] and Map an application to an existing CGRA.

CGRAs are different from multi-core architectures like Raw processor [18], TRIPS [19], CM-5 [20] ,etc., in that these are complete processors while CGRA is typically used as an accelerator to a processor to speed up loop execution. The turn of this century has seen a shift in the focus of CGRA research: most research is now on developing programming tools for them. Since CGRA designs vary widely, compilers are developed for a particular CGRA, for example, the ADRES [21] architecture comes with the DRESC [9], [22] compiler, the RaPiD [23] with SPR [24], and CGRA express [7] with modified EMS [25]. Attempts are being made to increase the independence of the mapping algorithm from the architecture. Instead of designing mapping algorithm for an architecture, algorithms now work for a basic description of the target architecture. However, the joint scheduling and operand routing problems are NP-complete. Therefore, existing compilation techniques, in an attempt to generate a good solution, take a long time [8], [25]–[27].

Compilation time is not a major concern for embedded systems, in which the application is compiled once, and then executed indefinitely. The DRESC compiler [9] uses simulated annealing to simultaneously solve the scheduling and routing problem. Park et al. [25], develop an edge-centric modulo scheduling (EMS) based mapping technique, which repeatedly searches for a route to connect the newly placed node to its predecessors. The runtime of their technique would be exponential, but is limited to polynomial time due to user-defined constraints. Grigorios et al. [27], propose a modulo scheduling based backtracking scheme to generate a good mapping. Compilation techniques by Ahn et al [8], [26] use an ILP inside the compiler to obtain better mappings.

The running time to generate a schedule for all CGRA compilation techniques is large. The main requirement for multithreading in CGRAs is the ability to re-map the kernels dynamically at runtime. This requires the scheduling process to be very fast, such that the performance benefit obtained by multithreading is not masked by the compile time involved. We do not know of any work that addresses this challenge.

One technique that allows kernels of several applications to be executed simultaneously is Polymorphic Pipeline Array (PPA) proposed by Park et al. [28]. The CGRA is divided into physically separate virtual cores, which are allocated to the kernels. However, all the kernels must be known at compile-time, and the entire schedule to execute all the kernels simultaneously is generated at compile-time. There is no change, or reconfiguration at runtime. In contrast, in our solution, threads can be invoked at runtime, and our technique will be able to shrink the existing schedules (at runtime) to make space for the new kernel. With our approach CGRAs can be used to accelerate kernels of multiple execution threads in the processor by dynamically allowing schedules to target conceptual subsections of the CGRA.

IV. Motivation for multithreading

As CGRAs are being used as accelerators to multi-core (and multithreaded) processors, we envision the need of multithreading on CGRAs. The capability of CGRAs to accelerate kernels of multiple threads running on the core at the same time, or multithreading, will improve the throughput and utilization of CGRAs for several reasons, including balancing memory requirements, communicating tasks, and most importantly, low CGRA utilization by a single loop kernel.

There are several reasons that CGRA utilization cannot be improved. For example, any recurrence cycle between loop iterations limits minimum achievable II [11]. In Figure 3(a), a DFG with a recurrence cycle between operation a and b along with its mapping on a 4x4 CGRA is shown. If this DFG is unrolled like in Figure 3(b) and mapped to a 4x4 CGRA as shown, it can be seen that II does not improve. In fact, the maximum utilization achievable by *any* larger size CGRA is at most 3 PEs.

Traditionally, II is the performance metric of a CGRA mapping, where performance is inversely proportional to II. However, II does not take into account throughput, an important metric in a multithreaded system. As II decreases, not only does execution time decrease, but the PE utilization also increases. This is illustrated as $I = N \times U \times II$, where I, N, and U are the number of instructions, the number of PEs, and the average utilization respectively. For a given application and CGRA, I and N are constant, making II and U inversely proportional. Therefore, to improve performance of a given

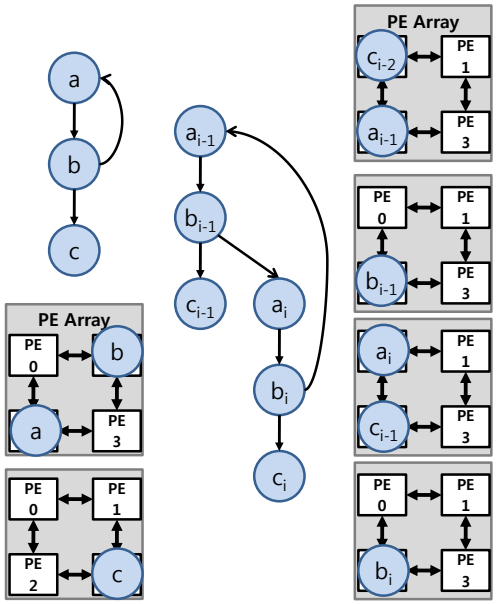


Fig. 3. (a) Original DFG and mapping with an II of 2. (b) Unrolled DFG mapped to a CGRA with an II of 4, creating an effective II of 2.

set of threads, the average utilization should be increased. Since a given kernel has a minimum constrained II, utilization can only be increased by running multiple kernels at a time. Performance can be seen as the number instructions executed per cycle, $IPC = \frac{\sum_{i=1}^{p} N_i \times II_i}{II_m}$ where $II_m = \max_{i=1}^{p} II_i$ and N_i is the number of PEs used by thread i. For simplicity, we assume that all threads have the same II. Thus,

$$\begin{aligned} IPC &= \frac{\sum_{i=1}^{p} N_i \times II_m}{II_m} \\ &= \sum_{i=1}^{p} \frac{N_i \times N}{N} \\ &= N \times U_a \end{aligned}$$

where U_a is the average PE utilization of the threads being ran. Therefore, we believe that CGRA support for multithreading is essential.

V. Key Idea in this Paper

The only way to accelerate multiple kernels simultaneously on a CGRA currently is to statically combine them into one, and then use kernel mapping techniques to map the combined kernel into a single schedule to be run on the CGRA. While this may be possible in extremely-embedded domains, this is not dynamic enough to be useful in general-purpose processors. Therefore, a key requirement of multithreading is that threads are to be compiled independently of each other. The generated CGRA schedules of different kernels can then be combined at runtime to be executed simultaneously.

The key idea in this paper is to enable multithreading on CGRAs based on space multiplexing. We dynamically shrink

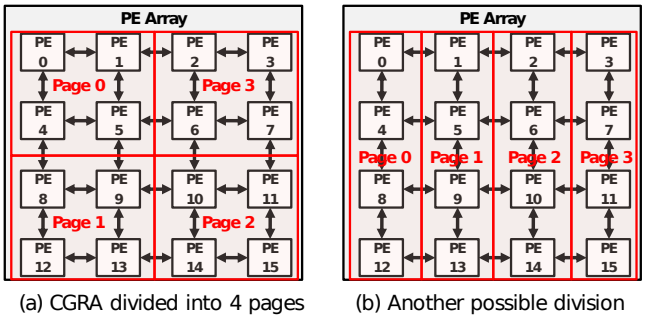


Fig. 4. Two acceptable ways to divide a 4x4 CGRA into 4 pages: 4 2X2 groups or 4 4X1 groups.

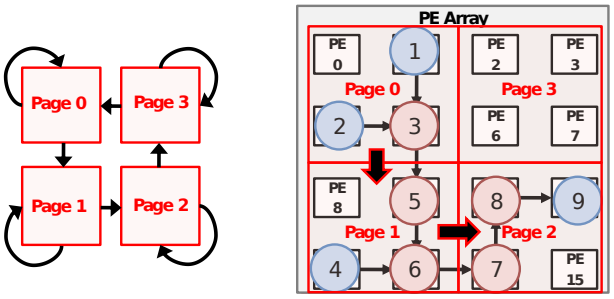
or expand the original schedule to use fewer or more PEs. When new threads are invoked, the existing kernel schedules can be shrunk to make space for the new thread, and when a thread leaves, the schedules of the running kernels can be expanded to better utilize the PEs. The key capability required for this technique is to take a given number of CGRA schedules, each using the whole CGRA, and combine them to create one schedule that uses the entire CGRA, but still satisfies all the dependencies of the input schedules, and a mechanism of making a transition between the mappings. The most important issue is that the melding of schedules and the transition into the newly combined schedule must be done quickly, as it is done at runtime. This is a difficult problem. A simpler subproblem of this problem is to take a single schedule and change the size of the CGRA that it uses. This problem is the same as that of mapping a kernel onto a CGRA (with fewer PEs). A scheduled graph just has some more *false* dependencies than the original graph. This task of kernel mapping/scheduling on CGRA has been shown to be NP-complete, and therefore time consuming. Consequently, naively using existing compilation techniques to re-map the schedules to a different sized CGRA at runtime will not work. We solve this problem by adding constraints on the original mapping, so that modifying the schedule later at runtime becomes easier (low-order polynomial time). An important concern is that the constraint on the mapping should maintain the performance (inversely proportional to II) of the original schedule.

VI. Our Approach

The two main aspects of our multithreading approach are i) constraints on the initial mapping and ii) a dynamic transformation to utilize a different size CGRA. In addition, we should also determine at what granularity we add these constraints and change the application mapping.

A. CGRA Paging

Our technique operates at a page level granularity. The CGRA structure is conceptually divided into *pages*, or symmetrically equivalent groups of PEs which allows page folding. Figure 4 shows two acceptable ways to divide a $4X4$ CGRA into 4 pages. While mapping and scheduling assigns a specific



(a) Ring topology of allowed dependencies

(b) Existing mapping satisfies the dependency constraints

Fig. 5. (a) Allowed dependencies between pages. (b) The original mapping already adheres to the mapping constraint. Page 1 depends on Page 0, and not vice versa. This is also the case for Page 2 and Page 1.

operation to be completed on a given PE at a given time, at the page level, mapping and scheduling refers to a set of operations to be completed on the PEs within a page at a given time. To expand or shrink the schedule, the operations are moved as groups to different pages; the mapping within a page remains the same, allowing a schedule to be shrunk to a minimum of one page and accelerated by at most N pages (the original number of pages compiled for). The number of PEs within each page and the total number of pages within a CGRA is a design parameter, determining the granularity of the transformation. Finally, it is important to note that no hardware modifications are required in the CGRA for paging. This is only conceptual, and required for compilation purposes. It should be noted that this method is different from tiling and clustering methods used in locally sequential globally parallel (LSGP) processors [29], [30] because the main purpose of paging in CGRAs is to be able to run multiple threads simultaneously, whereas in LSGPs, clustering is used to reduce mapping complexity of a single thread.

B. Compile-Time Constraints

The role of the compiler is to generate the initial schedule and mapping of the operations onto the PEs of the CGRA. This schedule will then be transformed to use fewer number of pages at runtime, when required. To simplify the transformation, we introduce two constraints to the compiler.

1) *Register Usage Constraints*: The compiler must use memory to store temporary variables that a PE may need. This simplifies moving the computation among pages. The local register file in the PEs will be used for the transformation. Many scheduling and mapping techniques [7]–[9], [25], [27], [31]–[35] do not effectively use local registers.
2) *Data Flow Constraints*: When we transform a schedule to use fewer number of pages, a requirement is not to break any data dependencies. To simplify the transformation, we allow data dependencies between pages to form a subset of ring topology. This is shown in Figure 5(a), where the data dependencies are such that the pages form a ring topology. This constraint simply means that PEs in Page 1 can depend on PEs from Page 1 or Page 0 of the previous time, but no other PEs. We demonstrate later in experiments that this constraint has little effect on the quality of mapping.

C. Dynamic Rescheduling

The output of the compiler is an efficient page-level mapping and scheduling of the application onto the whole CGRA, and if only one application is to be executed, no rescheduling of the mapping is required. However, when multiple applications are executed on the CGRA, each application must use only a portion of the CGRA resources. This re-partitioning is done at the page-level granularity. The total number of pages in the CGRA should be partitioned according to the number of threads executing. Our transformation algorithm reschedules the applications in low-order polynomial time to execute efficiently over fewer CGRA pages.

1) Problem Definition: Given an application mapped to the CGRA structure with the compile-time constraints, reschedule the application at page-level granularity to a CGRA with fewer number of pages.

Input: Application mapping P onto a CGRA with N pages. Suppose the Initiation Interval of the mapping is II_p. The mapping is specified as:

$$P = \{p_{(n,t)} : 0 \leq n < N, 0 \leq t < II_p\}$$

where $p_{(n,t)}$ represents the set of operations that will be performed on page n at time t. The constraint on the mapping is that the operations in $p_{(n,t)}$ are dependent through the interconnect (ring topology) from $p_{(n-1,t-1)}$ or through storage elements on the same page $p_{(n,t-1)}$.

Output: Application mapping Q onto M pages of the CGRA. The new mapping can be specified as:

$$Q = \{q_{(n',t')} : 0 \leq n' < M, 0 \leq t' < II_q\}$$

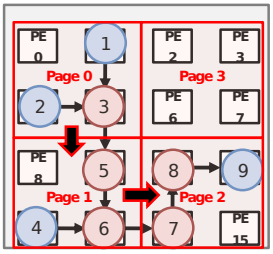
where $q_{(n',t')}$ represents the set of operations that will be performed on page n' at time t'.

Constraints: The first constraint is that no two pages in P must be mapped to the same page in Q. If $p_{(n,t)} \in P$ is mapped to $q_{(x',t')} \in Q$, we denote it by $p_{(n,t)} \rightarrow q_{(x',t')}$. Thus if $p_{(n_1,t_1)} \rightarrow q_{(x'_1,t'_1)}$, and $p_{(n_2,t_2)} \rightarrow q_{(x'_2,t'_2)}$, then if $n_1 \neq n_2$, then $x'_1 \neq x'_2$ and $t'_1 \neq t'_2$. The other constraint is that the mapping Q must not break any of the dependencies in P. Thus, for each n, t, if $p_{(n,t)} \rightarrow q_{(x_1,t_1)}$, $p_{(n-1,t-1)} \rightarrow q_{(x_2,t_2)}$, and $p_{(n,t-1)} \rightarrow q_{(x_3,t_3)}$, the constraints are:

- $(x_2 - 1 \leq x_1 \leq x_2 + 1)$ & $t_1 > t_2$
- $(x_3 - 1 \leq x_1 \leq x_3 + 1)$ & $t_1 > t_3$
- $x_1, x_2, x_3 < M$

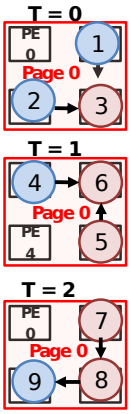
Objective: Clearly the objective of the mapping is to minimize the II of the mapping Q, II_q. If the II of the original mapping P is II_p, then $II_q \geq II_p \times \lfloor \frac{N}{M} \rfloor$, by resource constraints.

Time	Page 0	Page 1	Page 2	Page 3
0	p0,0	-	p2,0	p3,0
1	p0,1	-	p2,1	p3,1



(a) Page schedule with 4 pages

Time	Page 0
0	p0,0
1	p2,0
2	p3,0
3	p0,1
4	p2,1
5	p3,1



(b) Page schedule with 1 page (re-scheduled to page 0)

Fig. 6. (a)Table describes the time-schedule of the application on a 4-page CGRA. The mapping only uses 3 pages. (b)Rescheduling the mapping to execute on a 1-page CGRA. All three pages of the application are mapped to the same page (page 0) in the CGRA. Note that the mapping within the page also needs to be mirrored.

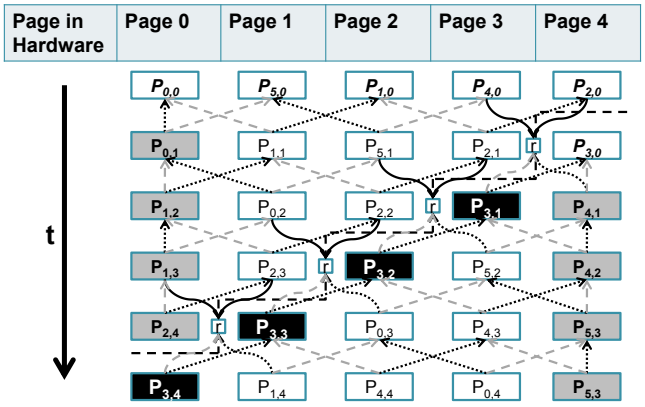


Fig. 7. Transformation from $N = 6$ pages to $M = 5$ pages. Italicized pages are from the first iteration of P. White pages were scheduled with *Case 1*, gray pages with *Case 2*, and black pages with *Case 3*. Dark arrows represent how same page dependencies are fulfilled, while gray arrows describe how previous page dependencies are fulfilled. Boxes with 'r' are registers used in that iteration.

D. PageMaster Transformation

Shrinking a schedule from N pages to a single page is the simplest: just execute the pages in order of dependency. Figure 6 shows the transformation from a 4-page schedule to a 1-page schedule. All pages are mapped onto the same page at different cycles, $Page0$. Note that when mapping page 1 onto $page0$, the internal page mapping also needs to change. The internal page mapping must ensure that the output of operation 3 is available for access by operation 5 in the next cycle. Fortunately, this transformation is simple and is achieved by simply mirroring the original mapping across the among-page dependency direction. For example, since the dependency between $Page0$ and $Page1$ is vertical, the mapping of $Page1$ must be mirrored along the horizontal axis. Additionally, $Page2$ is mirrored along the vertical axis.

To perform the runtime transformation, we introduce an algorithm named the *PageMaster Transformation*. It transforms a given schedule P at the page-level granularity to the CGRA structure with M pages ($M \leq N$). The key idea in the *PageMaster Transformation* algorithm is that given an N-page mapping of an application in the ring topology, every page mapped in the modified schedule (with M pages) is placed at a distance of at most 2 page columns (hops) from its dependent page in Q. Thus, all neighboring pages in P (defined as $p_{(n,t)}$ neighbors $p_{((n-1)\%N,t)}$ and $p_{((n+1)\%N,t)}$) are placed within two hops of each other in Q. This is because all pairs of neighboring pages share a common consumer and the furthest length any dependency can be transferred is two hops: one producer placed in column n, the other producer placed in column $n+2$, and the consumer placed in column $n+1$.

The dynamic transformation algorithm works in two phases: i) Reschedule the first iteration ($p_{(n,t)} \in P : 0 \leq n < N, t = 0$) of P to Q. ii) Place the remaining pages of P in Q maintaining the constraints and dependencies using Algorithm 1.

1) Schedule Initialization: To start, any arbitrary page $p_{n,0}$ must be placed in $q_{0,0}$. Its two neighboring pages must then be placed within the next two hops to maintain the two hop constraint. This will produce the following schedule:

- $p_{n,0} \rightarrow q_{0,0}$
- $p_{(n-1)mod(N),0} \rightarrow q_{1,0}$
- $p_{(n+1)mod(N),0} \rightarrow q_{2,0}$

Once this pattern is established, all subsequent pages are determined from these placements (i.e., it can be seen that $p_{(n-2)mod(N),0} \rightarrow q_{3,0}$ and $p_{(n+2)mod(N),0} \rightarrow q_{4,0}$). Indeed, as multiple rows of Q are filled with the first iteration of P, this same pattern holds, wrapping at the edges of the structure to form a complete and connected scheduling line. However, if the number of remaining pages from P to be scheduled in Q is not enough to complete a row, they are placed as tails in either the left-most or right-most column in Q such that $p_{n,0}$ is scheduled at an earlier time than $p_{n+1,0}$, for data dependency reasons. If a row is partially filled horizontally not using the tailing method, it will cause a non-optimal solution in the future as some locations in Q cannot be scheduled with a page from P. Figure 7 shows an example of how the first iteration would be placed in Q.

2) Filling the Rest of the Schedule: The rest of Q is filled individually for all remaining pages in P using Algorithm 1. To optimize runtime and placement, the algorithm is called for all pages in an iteration of P, and then for all pages in each subsequent iteration. The pages are placed in reverse order of how they are placed in the first iteration, such that the last page placed in the initiation phase is placed first in each iteration. This assures that `findDependencyColumns()` can be optimized to run in constant time by limiting the search space required (as locations of pages from a series can be estimated). The dependencies for a given page are the basis for where to place it in Q. There are three cases:

Data: d_1 is the column location of $p_{n-1,t-1}$
d_2 is the column location of $p_{n,t-1}$
t_1 is the next available time in the column being placed in after the time $p_{n-1,t-1}$ and $p_{n,t-1}$ have executed in Q

d_1, $d_2 \leftarrow findDependencyColumns()$
switch d_1 **do**
  case *($d_2 \pm 2$) /* Two hops apart */*
    $p_{n,t} \rightarrow q_{(d1+d2)/2,t1}$
  case *($d_2 \pm 1$) /* One hop apart */*
    if *(d_1 = 0 or d_2 = 0)* **then**
      $p_{n,t} \rightarrow q_{0,t1}$
    else if *(d_1 = M-1 or d_2 = M-1)* **then**
      $p_{n,t} \rightarrow q_{M-1,t1}$
    end
  case *(d_2) /* Zero hops apart */*
    if *(d_1 - 1 has less pages scheduled in it)* **then**
      $p_{n,t} \rightarrow q_{d1-1,t1}$
    else if *(d_1 + 1 has less pages scheduled)* **then**
      $p_{n,t} \rightarrow q_{d1+1,t1}$
    end
  end
end

Algorithm 1: PlacePage($p_{n,t}, Q$) outputs Q

1) *The dependencies are two hops apart*: The page is placed in the only applicable column at the earliest available time after which the dependencies have executed.
2) *The dependencies are one hop apart*: This case can only happen when a dependency is in column 0 or column $M-1$. Because these columns cannot be filled by the first most common case, the page is placed in column 0 or $M-1$ at the earliest available time after which the dependencies have executed.
3) *The dependencies are zero hops apart*: This case is a recurring case only for the pages placed as tails in the initialization phase. Because the case is recurring, the page is placed in column $d_1 - 1$ or $d_1 + 1$ at the earliest available time after which the dependencies have executed, depending on which column has fewer pages already scheduled in it.

3) Timing and optimality analysis: Algorithm `placePage` runs in constant time and is called for each page in P. There are $II_p \times N$ pages in P, and a constant time operation is performed for each. This allows the algorithm to run in low-order polynomial time with respect to the number of pages. An optimal $T(P) \rightarrow Q$ will have a page from P scheduled in every location in Q. The above algorithm will produce an optimal schedule, as it assures optimal placement with every iteration of P placed. For every iteration P, it is scheduled at the earliest time allowable in schedule Q. It should be noted that this optimality is for overall threads and it may increase a single thread's execution time.

E. Architecture Support Required

Our *PageMaster* transformation requires the CGRA architecture to have local registers. N rotating registers in each PE will ensure that the original mapping to the whole kernel can be shrunk to a single page when needed. Note that this is not a unreasonable architectural requirement, and most CGRA architectures e.g., MorphoSys [1], ADRES [3], offer both local register files and global storage mechanisms. Finally, the organization of PEs into pages is only conceptual, requiring no specialized hardware.

VII. Experimental Results

We performed experiments to verify two points: i) Minimal performance degradation by adding new compiler constraints, and ii) Performance improvement by enabling multithreading on CGRAs.

A. Minimal Performance Loss Due to Compiler Constraints

We first take a set of benchmarks and map them to a CGRA using an unmodified compiler to determine a baseline II_b. We then modify the compiler to follow our compile time constraints and compare this II to the baseline II_b. If average II remains within a few percentage points of II_b, we can conclude that there is no significant performance loss.

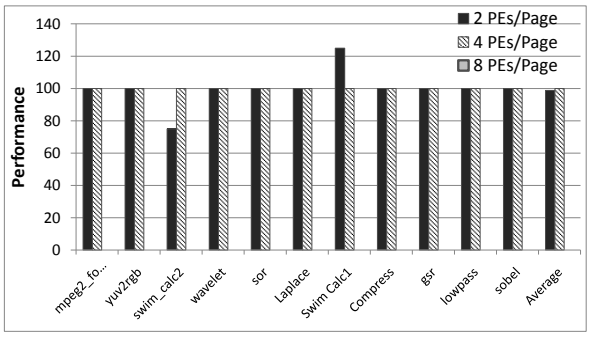
We use three CGRA sizes of 4x4, 6x6, and 8x8, with an architecture similar to that shown in Figure 1. We conceptually divide the CGRA into pages, using a page size of $2, 4$, and 8 PEs. We use a compiler based on the EMS mapping algorithm [25]. We experiment over a set of 11 benchmarks, including video decoding *e.g., mpeg, yuv2rgb*, highly parallel applications *e.g., Sor, Compress*, and filters *e.g., Gsr, Laplace, Lowpass, Swim, Sobel, Wavelet*, which represent applications commonly utilized in CGRAs.

Figure 8 shows that with paging constraints, the II of most applications is not affected considerably when page size is small. We see in Figure 8(a) that for a page size of 4, performance remains identical to the baseline mapping. A mapping for a 4x4 CGRA with 8 PEs a page was not generated, as there is not enough multithreading potential using only two pages. There is a slight performance degradation at the individual thread level for a page size of 2 PEs. We can conclude that performance will not be degraded with proper page size selection given the execution environment.

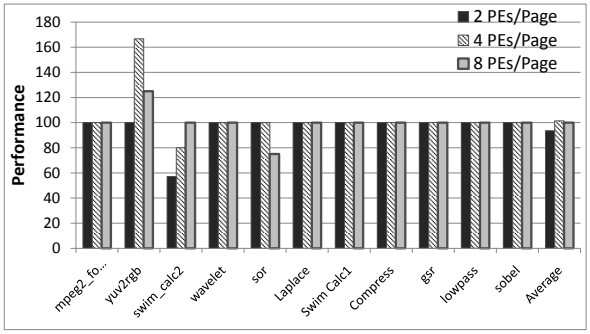
B. Enabling Multithreading

To show that the entire system performance increases when using a multithreaded CGRA, we simulate a system running under different loads with varying number of threads. We then demonstrate the need for a multithreaded CGRA. Here, we imagine a CPU running multiple threads, each having individual portions that need to be accelerated by the CGRA. We do this for two cases: (i) the case of a single-threaded, non-preemptive CGRA and (ii) the case of our multithreaded CGRA. The kernel/schedule that is to be ran on the CGRA comes from the set of benchmarks we compiled using both the original compiler and our multithreaded compiler constraints.

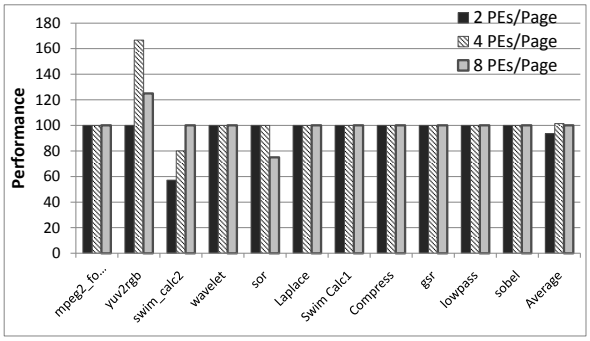
The OS is in charge of keeping track of currently running threads. When an additional thread is launched on the CGRA, the OS will transform the thread for the current environment



(a) Performance difference for a 4X4 CGRA



(b) Performance difference for a 6X6 CGRA



(c) Performance difference for a 8X8 CGRA

Fig. 8. Performance difference caused by paging constraints for different CGRA sizes. A performance of 100% indicates identical performance to original mapping, while greater than 100% is an increase in performance.

and transfer the thread into CGRA memory. Clearly, the time to transfer the thread into CGRA memory, including all its data, is greater than the time to execute the *PageMaster* algorithm; thus we assume that algorithm execution time is negligible. We assume that the instruction and data memory of the CGRA is large enough for the thread to be added.

1) Multithreading: Experimental Setup: In order to precisely evaluate performance improvement due to the multithreading technique, we perform simulation for different application for three separate CGRA needs:(1)*Low CGRA need* is 50%, (2)*Medium CGRA need*, is 75%, and (3)*High CGRA need* is 87.5% of a thread is scheduled to the CGRA. The reason we consider high CGRA need is to ensure that performance improvement is not only because of the processor multithreading. More precisely, because the relatively small percent of code is executed by the main processor in the high CGRA need case, multithreading improvement caused by the main processor becomes negligible in overall improvement according to the Amdahl's law [36].

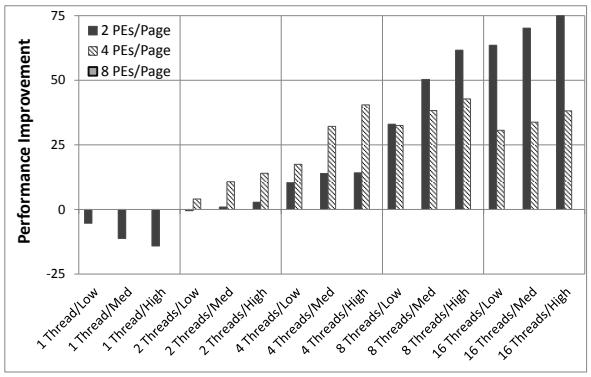
We run $1, 2, 4, 8,$ and 16 threads in parallel for each of the CGRA needs. Each thread is randomly and independently generated, where portions of the thread are either assigned to the processor or the CGRA. For portions assigned to the CGRA, the schedule that is ran is randomly chosen so as to not create bias towards any one kernel/schedule.

We perform the multithreading as follows: When only one thread is executing on the CGRA, it uses the entire CGRA. When another thread requests access to the CGRA, the thread using the most pages is decreased to use half as many pages and the new thread is resized to fit into the freed portion of the CGRA. The current thread is switched at a integer value of $II_p \times N/M$ to begin running on fewer pages and the other thread is scheduled to the remainder of the CGRA. When subsequent threads come, the current threads are shrunk as described, each time performing a transform from the original number of pages to the new desired number of pages (threads are expanded as other threads complete). In the cases where schedules do not use the entire CGRA, no transformation needs to be performed and the thread is simply scheduled to the unused portion of the CGRA.

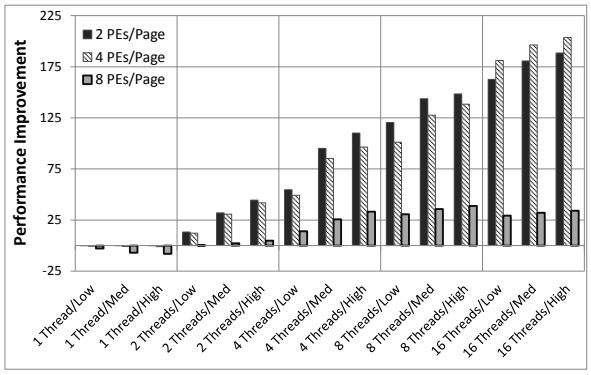
2) Analysis of Results: When the number of threads is low, paging constraints can degrade performance, as was the case in the 4x4 CGRA with a page size of 2 PEs, as shown in Figure 9. However, if a page size of 4 PEs was chosen, there would be no performance degradation. The trade-off is that there is less multithreading potential with a page size of 4 PEs on the 4x4 CGRA. Therefore, page size should be chosen according to the system needs. The case of the 4x4 CGRA is unique, as there are many more threads than pages, forcing threads to stall (the same bottleneck is observed for the 8 PE page size in the 6x6 CGRA). Thus multithreading performance is limited. However, as CGRA size increases and subsequently the number of pages available, multithreading performance greatly improves.

VIII. Conclusion

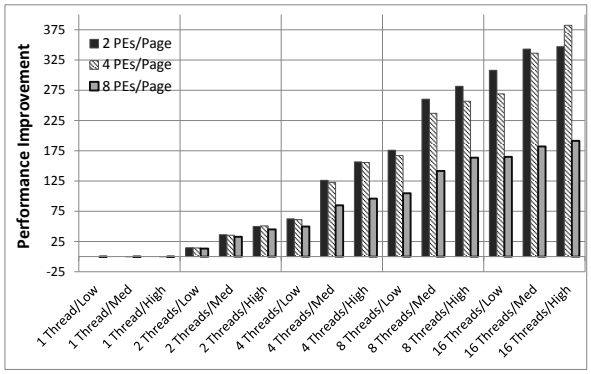
In the era of power efficient and performance hungry embedded applications, we observe that the CGRA is a strong contender to serve the growing needs. Compilation overheads for applications executed on the CGRA, and the inability to easily transform an application to execute over reduced CGRA resources dynamically, have prevented its use with multithreaded mainstream processors. In this work, we develop the first dynamic rescheduling framework that enables simultaneous execution of multiple applications on a CGRA, and also allow dynamically variation in the number of resources used by the threads running on the CGRA. We demonstrate through experiments over a wide range of applications the effectiveness



(a) Performance improvement in a 4X4 CGRA



(b) Performance improvement in a 6X6 CGRA



(c) Performance improvement in a 8X8 CGRA

Fig. 9. Performance improvement by adding multithreading support to CGRAs. The performance improvement in different CGRA sizes for different CGRA needs and thread numbers are shown for different page sizes is shown. A positive percentage indicates a performance increase.

and applicability of our framework to enable multithreading in CGRA. Our *PageMaster* transformation algorithm performs a runtime rescheduling in low-order polynomial time, and can be easily integrated with any existing or future mapping algorithms.

IX. Future Work

In this paper we experimented using different CGRA sizes and our *PageMaster* transformation algorithm is applicable to CGRA structures of any size (respecting the ring topology and interconnect constraints). Other page size configurations are also possible, so long as they remain symmetrical (to allow page folding). It can also be noted that the structure and implementation of our transformation algorithm is independent of the underlying mapping algorithm and therefore as a future work, we plan to study the impact of this framework integrated with other mapping algorithms and CGRA structures. In the future we hope to study the benefits of multithreading in regard to balancing memory requirements and thread communication.

Acknowledgments

This work was partially supported by funding from National Science Foundation grants CCF-0916652, CCF-1055094 (CAREER), NSF I/UCRC for Embedded Systems (IIP-0856090), Raytheon, Intel, SFAz and Stardust Foundation.

References

[1] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *Computers, IEEE Transactions on*, vol. 49, no. 5, pp. 465 –481, may 2000.

[2] "Intel Core2 Extreme Quad-Core Processor QX6000 and Intel Core2 Quad Processor Q6000 Sequence Datasheet." [Online]. Available: http://download.intel.com/design/processor/datashts/31559205

[3] S. Vassiliadis and D. Soudris, "ADRES & DRESC: Architecture and Compiler for Coarse-Grain Reconfigurable Processors," in *Fine- and coarse-grain reconfigurable computing*, D. S. Stamatis Vassiliadis, Ed., 2007.

[4] Y. Kim, M. Kiemb, C. Park, J. Jung, and K. Choi, "Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization," vol. 1. IEEE Computer Society, 2005, pp. 12–17.

[5] R. Hartenstein and R. Kress, "A datapath synthesis system for the reconfigurable datapath architecture," in *Proceedings of the 1995 Asia and South Pacific Design Automation Conference*. ACM, aug-1 sep 1995, pp. 479–484.

[6] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *Proceedings of the conference on Design, automation and test in Europe*. IEEE Press, 2001, pp. 642–649.

[7] Y. Park, H. Park, and S. Mahlke, "CGRA express: accelerating execution using dynamic operation fusion," in *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM, 2009, pp. 271–280.

[8] J. W. Yoon, A. Shrivastava, S. Park, M. Ahn, R. Jeyapaul, and Y. Paek, "SPKM: a novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures," in *Proceedings of the 2008 Asia and South Pacific Design Automation Conference*. IEEE Computer Society Press, 2008, pp. 776–782.

[9] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," vol. 150, no. 5, sept. 2003, pp. 255–61.

[10] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker, "Register allocation for software pipelined loops," in *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*. ACM, 1992, pp. 283–299.

[11] B. R. Rau, in *Proceedings of the 27th annual international symposium on Microarchitecture*. ACM, 1994, pp. 63–74.

[12] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "PipeRench: a co/processor for streaming multimedia acceleration," in *Proceedings of the 26th annual international symposium on Computer architecture*. IEEE Computer Society, 1999, pp. 28–39.

[13] D. C. Chen, "Programmable arithmetic devices for high speed digital signal processing," Ph.D. dissertation, EECS Department, University of California, Berkeley, 1992. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1992/2033.html

[14] J. Becker and M. Vorbach, "Architecture, memory and interface technology integration of an industrial/academic configurable system-on-chip (csoc)," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'03)*. IEEE Computer Society, 2003, pp. 107–112.

[15] T. Miyamori and K. Olukotun, "Remarc: reconfigurable multimedia array coprocessor," in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. ACM, 1998, pp. 261–.

[16] E. Mirsky and A. DeHon, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources," in *IEEE Symposium on FPGAs for Custom Computing Machines*, 1996, pp. 157–166.

[17] S. Y. Kung, *VLSI array processors*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987.

[18] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to software: Raw machines," *Computer*, vol. 30, no. 9, pp. 86 –93, Sep. 1997.

[19] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger, "Distributed Microarchitectural Protocols in the TRIPS Prototype Processor," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2006, pp. 480–491.

[20] W. D. Hillis and L. W. Tucker, "The CM-5 Connection Machine: a scalable supercomputer," *Commun. ACM*, vol. 36, pp. 31–40, November 1993.

[21] B. Mei, F.-J. Veredas, and B. Masschelein, "Mapping an H.264/AVC decoder onto the ADRES reconfigurable architecture," in *Field Programmable Logic and Applications, 2005. International Conference on*, aug. 2005, pp. 622 – 625.

[22] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "DRESC: a retargetable compiler for coarse-grained reconfigurable architectures," in *Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on*, dec. 2002, pp. 166 – 173.

[23] C. Ebeling, D. Cronquist, P. Franklin, J. Secosky, and S. Berg, "Mapping applications to the RaPiD configurable architecture," in *FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, apr 1997, pp. 106 –115.

[24] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "SPR: an architecture-adaptive CGRA mapping tool," in *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2009, pp. 191–200.

[25] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM, 2008, pp. 166–176.

[26] M. Ahn, J. W. Yoon, Y. Paek, Y. Kim, M. Kiemb, and K. Choi, "A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures," in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, 2006, pp. 363–368.

[27] G. Dimitroulakos, S. Georgiopoulos, M. D. Galanis, and C. E. Goutis, "Resource aware mapping on coarse grained reconfigurable arrays," *Microprocess. Microsyst.*, vol. 33, pp. 91–105, March 2009.

[28] H. Park, Y. Park, and S. Mahlke, "Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009, pp. 370–380.

[29] A. Darte, "Regular partitioning for synthesizing fixed-size systolic arrays," *Integration, the VLSI Journal*, vol. 12, no. 3, pp. 293 – 304, 1991.

[30] A. Darte, R. Schreiber, B. Ramakrishna Rau, and F. Vivien, "A constructive solution to the juggling problem in processor array synthesis," in *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, 2000.

[31] G. Dimitroulakos, M. Galanis, and C. Goutis, "A compiler method for memory-conscious mapping of applications on coarse-grained reconfigurable architectures," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, 2005, p. 4 pp.

[32] J. eun Lee, K. Choi, and N. Dutt, "Compilation approach for coarse-grained reconfigurable architectures," *Design Test of Computers, IEEE*, vol. 20, no. 1, pp. 26 – 33, jan-feb 2003.

[33] A. Hatanaka and N. Bagherzadeh, "A Modulo Scheduling Algorithm for a Coarse-Grain Reconfigurable Array Template," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007, pp. 1 –8.

[34] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins, "Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture: A Case Study," in *Proceedings of the conference on Design, automation and test in Europe*. IEEE Computer Society, 2004, pp. 1224–1229.

[35] H. Park, K. Fan, M. Kudlur, and S. Mahlke, "Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. ACM, 2006, pp. 136–146.

[36] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, pp. 483–485.