Recurrence Cycle Aware Modulo Scheduling for Coarse-Grained Reconfigurable Architectures

Taewook Oh Bernhard Egger
Samsung Advanced Institute of Technology
Giheung, Republic of Korea
{taewook.oh, bernhard.egger}@samsung.com

Hyunchul Park Scott Mahlke
Advanced Computer Architecture Laboratory,
University of Michigan
Ann Arbor, MI, USA
{hcpark,mahlke}@umich.com

Abstract

In high-end embedded systems, coarse-grained reconfigurable architectures (CGRA) continue to replace traditional ASIC designs. CGRAs offer high performance at a low power consumption, yet provide flexibility through programmability. In this paper we introduce a recurrence cycle-aware scheduling technique for CGRAs. Our modulo scheduler groups operations belonging to a recurrence cycle into a clustered node and then computes a scheduling order for those clustered nodes. Deadlocks that arise when two or more recurrence cycles depend on each other are resolved by using heuristics that favor recurrence cycles with long recurrence delays. While with previous work one had to sacrifice either a fast compilation speed in order to get good quality results, or vice versa, this is not necessary anymore with the proposed recurrence cycle-aware scheduling technique. We have implemented the proposed method into our in-house CGRA chip and compiler solution and show that the technique achieves better quality schedules than schedulers based on simulated annealing at a 170-fold speed increase.

Categories and Subject Descriptors C.3 [*Special-Purpose and Application-Based Systems*]: Real-Time and Embedded Systems; D.3.4 [*Processors*]: Code Generation and Retargetable Compilers

General Terms Algorithms, Design, Performance

Keywords Coarse-grained Reconfigurable Architectures, Placement and Routing, Software Pipelining.

1. Introduction

The trend for more complex and diverse mobile applications require today's portable devices to deliver high performance while maintaining a low energy consumption. Traditionally, ASICs have been used to meet these demands. However, ASICs have two major drawbacks: one is the lack of flexibility in functionality, and the other one are the high non-recurring costs involved in designing them. These shortcomings become more and more crucial due to the ongoing convergence of diverse functionality such as audio/video decoding, 3-D graphics rendering, data streaming, etc., as well as the increasing time-to-market pressure of portable electronic devices.

Coarse-grained reconfigurable architectures (CGRA) have recently emerged as an attractive alternative to ASIC-based solutions. The programmability of CGRAs combined with their abundant computing power and low power consumption perfectly meet the demand for flexibility, high performance, and low energy consumption. CGRAs consist of a large number of functional units (FU) and routing resources that interconnect the FUs. FUs are not necessarily homogeneous, i.e., some can implement memory operations while others support multiplication or SIMD operations. The FUs of a CGRA are typically arranged in a grid pattern. In addition to FUs and interconnections, CGRAs also contain instruction, data, and configuration memory, and bus connections for off-chip memory accesses. Compared to FPGAs, CGRAs sacrifice gate level reconfigurability for fast configuration, high operating frequency, and low power consumption.

An intelligent CGRA scheduler is essential to utilize the abundant computing power of the FUs in a CGRA effectively. Since not all FUs are directly connected to the central data register file and bypass logic does not exist between every two FUs in the grid, a CGRA scheduler computes a schedule that not only considers the placement of the operations but also the routing of the data operands. Thus, the scheduler is the critical factor that determines the overall performance of the CGRA.

To exploit CGRAs effectively modulo scheduling [15] that applies software pipelining of loops is widely used. DRESC [11] is a modulo scheduling framework for CGRAs based on simulated annealing. The DRESC framework generates an initial schedule that considers only operation dependencies but ignores resource overuse. In a second step, the scheduler randomly moves operations over the FUs until a valid schedule is found. This approach produces good schedules but the simulated annealing algorithm can take too much time (on the order of days for somewhat complex loops) to be used in a production compiler. To overcome that edge-centric modulo scheduling (EMS) [14] has been proposed. Most modulo scheduling techniques concentrate on the placement of operations, EMS focuses on the routing of the operands. The EMS framework requires far less time to find a schedule (on the order of minutes). However, the quality of the schedules does not meet that of the DRESC framework.

LCTES'09, June 19–20, 2009, Dublin, Ireland.

Our analysis shows that the major reason for the degradation of the quality of a schedule is caused by speculative scheduling of operations that belong to a recurrence cycle, i.e., a set of operations that depend on the outcome of computations from a previous loop iteration. In the EMS framework, the scheduling slot of an operation depends on the schedule of the operands' producers. The data-flow graph of a recurrence forms a cycle, thus some operations need to be scheduled before their producers have been placed. Consequently, some operations are scheduled with only partial information of their producers' schedule. This may impose additional constraints on the possible schedule times of other operations and thereby prevents EMS from finding a good overall schedule. In fact, not only the EMS technique, but all prevalent CGRA scheduling techniques that are based on prioritized topological ordering suffer from the same problem.

In this work, we propose a scheduling technique that is aware of data dependencies caused by inter-iteration recurrence cycles. In this technique, operations in recurrence cycles are clustered and treated as a single node. As soon as all producers of the clustered node have been scheduled, the operations of the recurrence cycle are handled.

The contributions of this work are as follows:

- We propose a recurrence cycle aware modulo scheduling technique. Our technique employs the fast edge-centric approach to place and route operations and operands, but is aware of recurrence cycles to assign a proper scheduling order for operations and thereby remove unnecessary scheduling constraints.
- We suggest an architectural modification to further improve the scheduling quality. Our analysis shows that the scheduling quality can be improved significantly with only a small modification to the architecture. We describe the necessary modifications and address their impact.

We have implemented the proposed method into our in-house CGRA chip and compiler solution which is used for several next-generation multimedia devices. The experimental results show that the proposed technique achieves better quality schedules than the scheduler based on simulated annealing at a 170-fold speed increase.

This paper is organized as follows. Section 2 gives an overview of related work. In Section 3 the background of and the motivation for this work are introduced. Section 4 explains the proposed scheduling technique in detail. Section 5 proposes architectural modifications that further improve the scheduling quality. The experimental results of the proposed scheduling technique and the effect of the architecture modification are discussed in Section 6. Section 7, finally, concludes the paper.

2. Related Work

Recently, several CGRAs have been proposed, each with its own characteristics. Hartenstein [6] provides a good summary of the characteristics for the following architectures: ADRES [12], MorphoSys [10], PipeRench [5], and RaPid [4].

Along with the architectures various techniques have been proposed for computing efficient schedules for CGRA. One class of algorithms tackles the problem by scheduling a single iteration of the loop on the CGRA. To exploit inter-iteration parallelism the loop body is first unrolled and then scheduled. Both [1] and [16] minimize the utilization of computational resources on the CGRA. Ahn *et al.* [1] divide the problem into coveting, partitioning, and laying-out. They first group operations that are to be scheduled together and then place them onto the FUs. Finally, they build the interconnection network between the FUs that minimizes the critical path. Yoon *et al.* [16] apply a technique named SPKM (split-push kernel mapping) that is based on the split&push algorithm used in graph drawing. Lee *et al.* [8] propose an approach that generates a pipelined schedule for the operation tree. Their technique focuses on computing a schedule that shares memory operations between successive iterations to reduce the effect of the memory bottleneck. Bansal *et al.* [2] introduce an algorithm that takes the number and the delay of interconnections into account. Their cost function considers the priority and affinity of operations based the connections between FUs.

A second class of algorithms focus on modulo scheduling. Mei *et al.* [11] propose an algorithm that uses simulated annealing. Their approach computes efficient schedules, however the compilation time lies in the order of days and is thus not well suited for a production environment. The approach of Hatanaka *et al.* [7] is similar to the work presented in [11]. They implement a resource-aware placement scheme to shorten the scheduling time. Even though they achieve a speedup of about 65% the resulting compilation time is still too long to use the algorithm in practice. Park *et al.* [13] introduce a scheduling method that adopts classical graph embedding methods. It leverages graph embedding to draw the data-flow graph of a loop body onto a three-dimensional graph representing the CGRA. In [14], the same authors propose the edge-centric modulo scheduling (EMS) framework that focuses on routing data operands rather than data operations. Their work shows that the EMS approach achieves a 10-13% improvement in performance and a 27-46% reduction in compilation time compared to a node-centric modulo scheduler.

Even though the work of Llosa [9] is not intended for CGRA, their swing modulo scheduling handles operations in recurrence cycles in a similar way to our technique. It handles operations in recurrence cycles before other operations. In [9], recurrence cycles are ordered based on their recurrence delay, while in our approach topological sorting is used to determine the scheduling order.

The technique presented in this work is based on an edge-centric approach. By focusing on recurrence cycles it maintains the fast compilation time of edge-centric modulo schedulers, but achieves a scheduling quality comparable to a modulo scheduler with simulated annealing.

3. Background and Motivation

3.1 Coarse-Grained Reconfigurable Architectures

A coarse-grained reconfigurable architecture (CGRA) consists of a large number of functional units (FU) each of which can execute word-level operations. The FUs communicate with each other through an interconnection network. When mapping a loop onto the CGRA, the compiler exploits the instruction level parallelism to utilize as many of the FUs as possible.

The FUs are not necessarily homogeneous. While usually all of them provide basic arithmetic operations, some support multiplication, others memory operations. Mixed-width designs are also possible with some of the FUs supporting wide SIMD (single instruction multiple data) operations. There are various kinds of topologies and sizes of FUs, from "4-in-a-row" to 8x8 grid architectures. Fig. 1 presents an example of a CGRA consisting of 16 FUs arranged in a 4x4 grid.

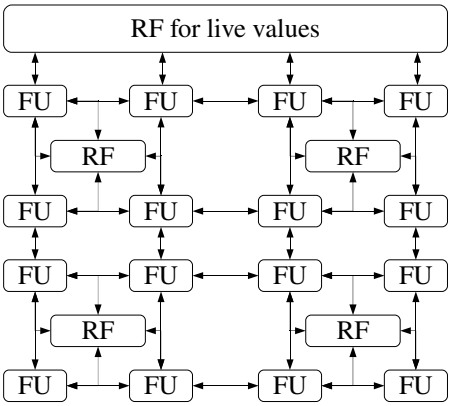

Figure 1. 4x4 CGRA architecture

There exist various kinds of interconnection networks between the FUs such as mesh-like networks, shared-bus architectures, shared register files between some FUs, or a mixture thereof. Note that an FU can also be used as a routing resource (i.e., forwarding a value), not only as a computational resource. In Fig. 1, the 16 FUs form a mesh network with their neighboring FUs and a group of four FUs share one local register file (RF) containing the live values. Live values (or live operands) are input or output values that are passed to the loop. These values are usually passed via register files to reduce the pressure on the memory bus. Generally, the existence of a single-cycle operand routing path between any two FUs is not guaranteed. Thus, the scheduler must explicitly determine the set of routing resources for each operand.

A special memory holds the configuration for every configurable unit for each cycle of a loop. In the case of an FU, the configuration memory holds the operation for that specific FU at the given loop cycle. The schedules for several loops are placed in the configuration memory.

3.2 Edge-Centric Modulo Scheduling

Modulo scheduling [15] is a form of software pipelining of loops to exploit the parallelism of the CGRA. A modulo scheduler overlaps different iterations of a single loop and then exploits instruction- and loop-level parallelism. The primary goal of modulo scheduling is to minimize the *initiation interval* (II), the number of cycles between the start time of successive iterations. In other words, one loop iteration takes II cycles (ignoring the loop prologue and epilogue). Therefore, minimizing the II directly leads to an increased throughput. Each iteration of a loop consists of $\lceil \textit{schedule time of single iteration}/II \rceil$ stages, thus a schedule of II cycles consists of the instructions of successive iterations in different stages. The modulo scheduler first calculates the initial (and lowest possible) II by taking the maximum of the *resource-constrained lower bound* (ResMII) and the *recurrence-constrained lower bound* (RecMII). It then tries to find a valid schedule within that minimal II. If no valid schedule can be found for the given II, the scheduler increments the II by one and tries again. This process is repeated until a valid schedule has been found.

What decisively distinguishes EMS [14] from other modulo scheduling-based CGRA schedulers is that it focuses on the routing of operands rather than the placement of operations. In the traditional node-centric approach, the scheduler first places producer and consumer operations and only then tries to find a routing path between them. In the edge-centric approach, however, the scheduler explores the routing resources near the FU where the producer operation is placed and expands the routing path along available resources. The scheduler places the consumer operation as soon as the expanded routing path meets a resource that supports the destination operation. That is, a node-centric approach first places the operations and then searches for a routing path while an edge-centric approach first checks the routability and then places the operation.

The edge-centric approach has two advantages over the node-centric approach: (a) a reduction of the scheduling time and (b) a performance improvement of the schedules.

3.3 Motivation for an Intelligent Ordering Scheme

Although EMS enables fast and efficient scheduling through placement and routing of each operation and its operands locally, it does not provide a solution for the global problem on how to determine the scheduling order of the operations. Existing EMS frameworks order operations based on their height in the data-flow graph (DFG). For operations that are part of a recurrence cycle, the scheduling time is assigned speculatively, that is, it is determined without having complete knowledge on the schedules of its producers. This may impose unnecessarily strict constraints on the scheduling process and lead to a degradation of the scheduling quality, namely, an increased II of the final schedule.

There are two types of constraints that must be considered during the scheduling process: (a) the resource constraint. Scheduling fails when all adjacent routing resources are already occupied during the routing of a data-flow edge, or when no more FUs are left that can support the operation to be scheduled. (b) the time constraint. Whenever a consumer operation is placed before one of its producer operation, its scheduling time bounds the latest possible scheduling time of the producer operations, and vice versa.

Operations that are part of a recurrence cycle have an innate time constraint since all operations in the recurrence cycle must be scheduled within II cycles. Therefore, once the scheduling time of one operation is fixed, the scheduling times of all other operations in the same recurrence cycle are bounded. Furthermore, these bounds also affect the maximum schedule time of producers, or the minimal schedule time of consumers that lie outside the recurrence cycle. Unnecessary restrictions on operations outside the recurrence cycle can be eliminated if operations in the recurrence cycle are properly scheduled. Simple topological ordering of the DFG cannot determine an adequate scheduling time of an operation if one of its incoming edges forms a recurrence cycle, hence one of them has to be scheduled without having full knowledge about the scheduling time of its producer. Usually, the loop head operation of a recurrence cycle is scheduled by ignoring the incoming recurrence back-edges.

Fig. 2 shows an example of the drawbacks of speculative scheduling of operations in a recurrence cycle. Fig. 2(a) shows the DFG of the loop to be mapped onto the CGRA, and Fig. 2(b) depicts the target architecture consisting of four FUs and one shared rotating register file [3]. We assume that the RF has four entries and that routing an operand through the RF takes an additional cycle compared to direct FU-to-FU routing.

There are two recurrence cycles present in the DFG: one is formed by the back-edge from operation 8 to operation 3 and includes the loop head, operation 3, plus operations 5 and 8. The second recurrence cycle created by the back-edge from operation

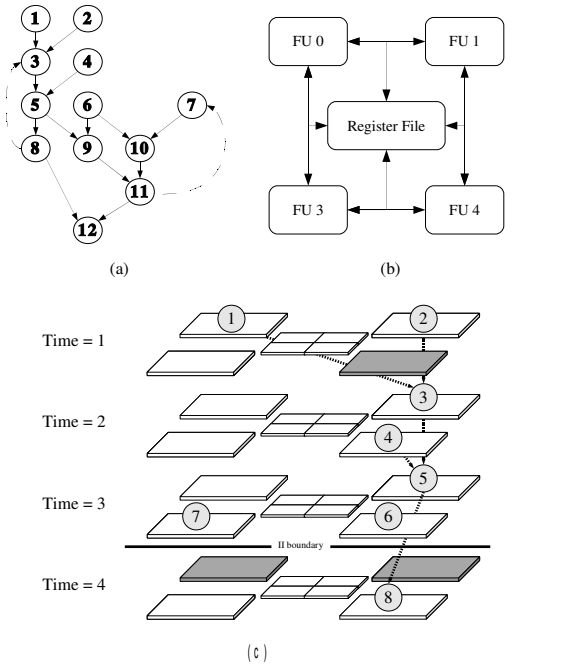

Figure 2. (a) data-flow graph (b) simplified target architecture (c) snapshot of the schedule when the scheduling process is stuck

11 to the loop head, operation 7, includes operations 7, 10, and 11. Operations 1, 2, 4, and 6 have no incoming edges and thus have the highest scheduling priority. Additionally, operation 7 also has the highest priority since the recurrence back-edge is the only incoming edge of operation 7. The scheduling order between the operations with the same priority is not of big importance, but it should be noted that once operation 7 is placed then the maximum scheduling times of operations 10 and 11 are also limited. Fig. 2(c) shows the partial schedule with $II = 3$ after operations 1 to 8 have been scheduled. A number i on a slot signifies that operation i has been mapped onto the corresponding FU. Shaded slots represent FUs that are occupied due to the modulo constraint.

For the given partial schedule the scheduler cannot progress because there is no valid scheduling slot for operation 9: since operation 7 is scheduled at $t = 3$, the operations 10 and 11 should be scheduled at $t = 4$ and $t = 5$, respectively, due to the innate timing constraint imposed by the recurrence cycle. Operation 9 must be scheduled on the only free slot at $t = 4$, because its producer, operation 5, has already been scheduled at $t = 3$ and there is no route available that forwards the result of operation 5 to operation 9 in one cycle. One might ask why operation 7 is scheduled at $t = 3$. This is because its earliest schedulable time is $t = 3$ if operations 1 and 2 are scheduled at $t = 1$ and operation 11's earliest schedulable time has been calculated as $t = 5$. It is reasonable to try to schedule operation 7 at the earliest schedulable time because if it is scheduled at a later time then the length of the overall schedule increases and additional routing resources are consumed.

However, this estimated earliest schedulable time alone is not sufficient to determine the scheduling time of operation 7. The assumption that operations 3, 5, and 9 will be scheduled at $t = 2, 3, and 4$, respectively, might be wrong because the routing between two operations may take more than one cycle on CGRA, thus it might not be possible to schedule operation 7 at $t = 3$. The actual scheduling times of these operations are unknown until they have been scheduled. However, with simple topological ordering, operation 7 gets scheduled earlier than the producer of operation 11 (i.e., operation 9) whose scheduling time is strictly bounded by the scheduling time of operation 7. Therefore, we have to rely on speculative scheduling and fail to find a valid schedule.

To overcome this problem, Park *et al.* [14] suggest a stage reassignment scheme. It lowers the scheduling time of operations in the recurrence cycle along the time axis by multiples of II. Since operations in the modulo scheduling framework are repeated every II cycles, such an adjustment neither changes the resource occupancy nor affects the operations outside the recurrence cycle. However, it may consume too many routing resources since it widens the distance between producers and consumers more than necessary, especially for loops with large IIs. Moreover, it stretches out the scheduling length of one single iteration which seriously affects performance for loops with small iteration counts.

The root cause of these obstacles is the improper scheduling of operations in recurrence cycles. This paper introduces a new scheduling algorithm focusing on recurrence cycles and proposes an architecture modification that leverages the efficiency of the scheduler.

4. Recurrence Cycle Aware CGRA Scheduling

4.1 Core Idea

As shown in the previous section, speculative scheduling of operations in a recurrence cycle leads to a suboptimal schedule. To conquer this problem we propose an intelligent operation ordering scheme that considers the relationship between recurrence cycles.

The proposed scheme treats recurrence cycles in the DFG as a single unit. Instead of scheduling each operation individually, the algorithm first groups all operations in a recurrence cycle into a clustered node. The operations of a clustered node are then scheduled together. Fig. 3 and 4 show the DFG before and after clustering of the recurrence cycles. Circles with a double line in Fig. 4 represent clustered nodes.

This transformation is important because it solves the problem of redundant time constraints imposed on producers outside the recurrence cycle (see previous section). Without clustering recurrence cycles, redundant time constraints are imposed since it is impossible to establish a strict topological order on operations that form a cycle in the DFG. Clustering recurrence cycles transforms the DFG into an acyclic graph, and an exact height-based topological order can be computed. All producers of the clustered recurrence cycle are now scheduled first even though some of them have a lower height than some operations in the recurrence cycle. Since all producers of the recurrence cycle are scheduled before any operation within the recurrence cycle, scheduling it is no longer speculative.

In addition, in the proposed scheme clustered nodes have priority over single nodes. That is, as soon as the scheduler has scheduled all producers of a clustered node, that node is immediately scheduled regardless of its height in the clustered DFG. Because the innate time constraint of recurrence cycles makes scheduling tricky, it is better to schedule the recurrence cycles as early as possible when still relatively many resources are unoccupied.

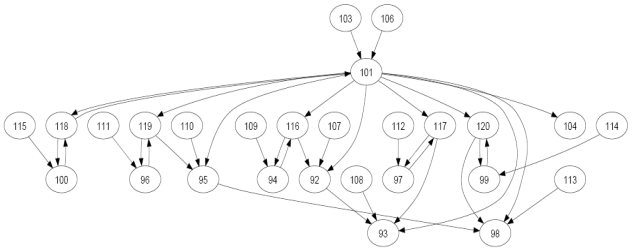

Figure 3. Data-flow graph before clustering

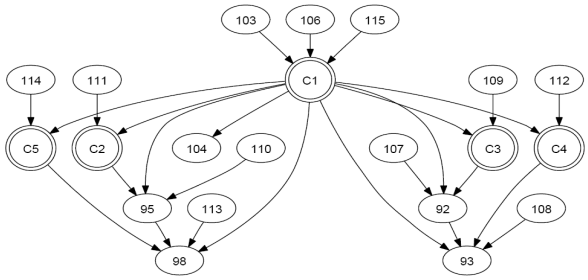

Figure 4. Data-flow graph after clustering

Strictly speaking, the clustered DFG may still contain cycles in case that there exist mutual data dependencies between two recurrence cycles. Section 4.3 explains how to deal with such situations in detail.

4.2 Implementation

This section describes the implementation of the proposed scheduling algorithm. Fig. 5 shows a flowchart of the implementation.

First, the scheduler computes the clustered DFG. Then, it selects the clustered nodes according to their priority and schedules them one by one. After all clustered nodes have been scheduled, the remaining nodes are handled.

When choosing a clustered node to be scheduled, nodes which have all of its clustered producers scheduled have the highest priority. In case that there is more than one clustered node which satisfied said condition, the scheduler picks the one with the largest incoming tree, i.e., the largest set of operations whose results contribute to the generation of the source operands of the clustered node. This is because a larger number of operations in the incoming tree requires the scheduler to consider more operations before it can schedule the recurrence cycle, and these operations should be scheduled when most computing resources are still unoccupied. It is possible that none of the clustered nodes meets the requirement that all clustered nodes on its incoming tree must have been scheduled. This situation occurs when two or more loops are data dependent on each other, and the scheduler cannot determine a scheduling order between them. In this case the scheduler uses a heuristic to resolve the deadlock (see Section 4.3).

The scheduling of a clustered node can be divided into three major steps: (1) scheduling of the incoming tree, (2) calculating the earliest scheduling time, and (3) scheduling the nodes of the clustered node.

When scheduling the incoming tree, it is possible that not all nodes can be scheduled if mutual data dependencies exist between two clustered nodes. In that case, operations that consume values produced by yet unscheduled clustered nodes are removed from the incoming tree. The remaining instructions are scheduled with a

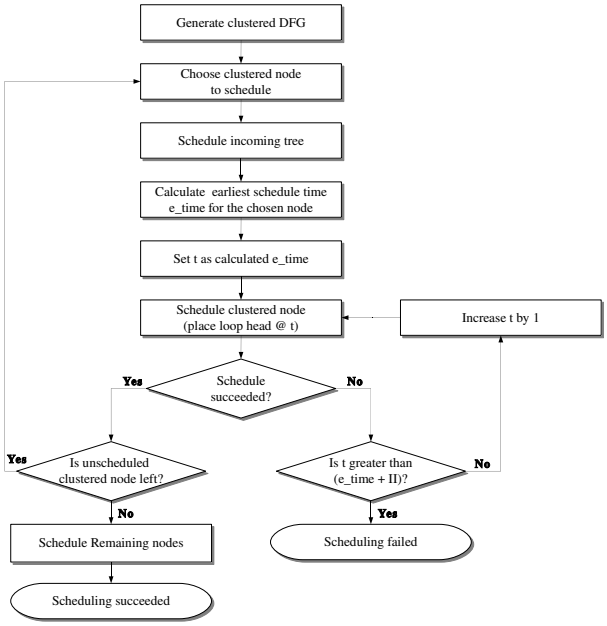

Figure 5. High-level flow chart of the proposed scheduling algorithm

height-based priority scheme (note that there exist no cycles in the incoming tree). The cost function used for scheduling the tree tries to avoid valuable (i.e., scarce) resources rather than reducing the routing delay. The cost function considers resources that can access the register file containing the live values particularly valuable since values that are both live-in and live-out form a recurrence cycle in the DFG. Those resources should be kept free for providing access to the live values when the recurrence cycle accessing them is scheduled.

Once the incoming tree has been scheduled completely, the clustered node itself is handled. The earliest schedulable time of the loop head of the recurrence cycle is computed with the following equation

$$e\_time(C) = \max_{o \in C}(ASAP(o) - dist(h, o))$$

e_time is calculated by taking the maximum ASAP (*as soon as possible*) time of all operations o that are part of the recurrence cycle C plus their distance from the loop head h. The distance $dist(x, y)$ of two operations denotes the distance of x from y in the DFG, in other words the latency of the operations on the path from x to y. The ASAP time for each operation is computed as follows

$$ASAP(o) = \max_{p \in prod_p(o)}(time(p) + latency(p))$$

where o denotes an operation in the clustered node, and $prod_p$ is the set of the placed producers of o. $time(p)$ stands for the schedule time of p, and $latency(p)$ expresses the latency of the operation p.

Next, the loop head operation is placed on an available resource at the calculated earliest schedulable time. After the loop head operation has been placed, the recurrence edge coming out of the loop head operation is scheduled along with its consumer. Following this procedure, all operations in the recurrence cycle can be scheduled based on the schedule of its producers. While scheduling each operation, edges from the incoming tree to the operation are also sched-

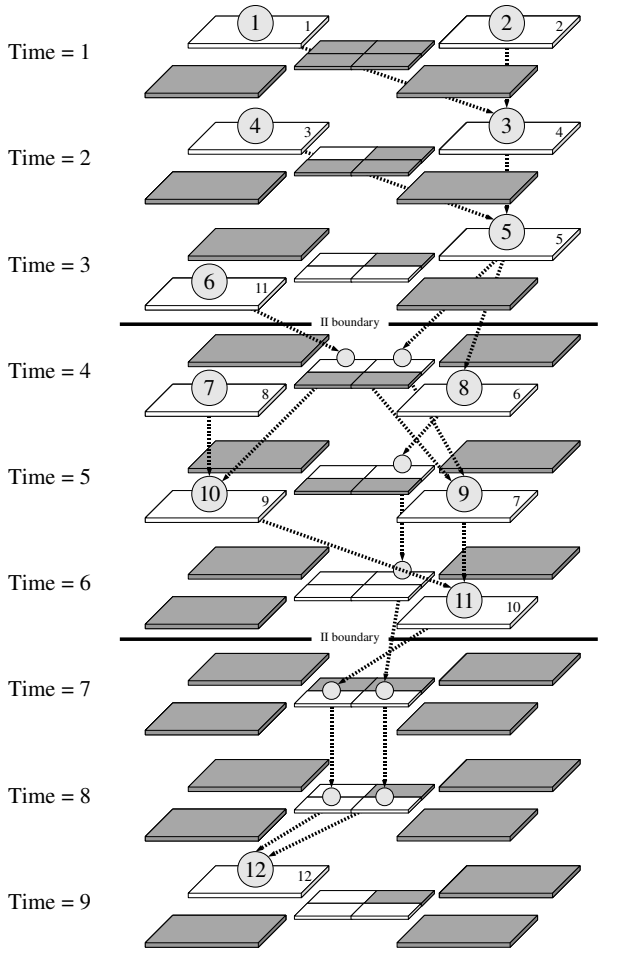

Figure 6. Successful schedule for the DFG shown in Fig. 2(a)

uled. If any of the edges or operations fails to be placed, the scheduler aborts the incomplete schedule and starts over by placing the loop head operation on another resource at the earliest schedulable time. If all possible slots have been tried unsuccessfully, the scheduler increases the loop head's schedule time by one cycle. This process is repeated until a valid schedule is found or the loop head's schedule time reaches $e\_time + II$. We limit the maximum schedule time to $e\_time + II$ because otherwise the distance between operations in the incoming tree and the recurrence cycle itself becomes too long and routing the operands consumes too many resources. The separation of scheduling the incoming tree and the recurrence cycle itself makes aborting a failed partial schedule easy. While scheduling the recurrence cycle, the scheduler uses a different cost function that strongly penalizes routing delay since all operations of the recurrence cycle must be scheduled within II cycles.

Once all clustered nodes and operations between them have been scheduled, the scheduler handles the remaining operations. These operations do not have any kind of time constraints but it is more difficult to find a route between a consumer and its producer because most routing resources are already occupied.

Fig. 6 depicts a successful final schedule of the DFG shown in Fig. 2(a) computed with the proposed scheduling method. The numbers in the top-right corners of the resources indicate the scheduling order of each operation. The routing paths of recurrence back-edges are not shown. The DFG from Fig. 2(a) contains two recurrence cycles: the first one, denoted $C1$, is formed by the operations 3, 5, and 8, and the second one, denoted $C2$, contains the operations 7, 10, and 11. Since $C2$ has a data dependency on $C1$, the scheduler handles the latter first. To schedule $C1$, the scheduler first attacks the operations of $C1$'s incoming tree, namely operations 1, 2, and 4. Then, the scheduler calculates $C1$'s earliest schedulable time as $e\_time(C1) = 2$, and then schedules the recurrence cycle's loop head operation 3 at that time. Based on operation 3's schedule, the remaining operations in $C1$ are scheduled. Next, $C2$ is handled. Since operation 9 is part of $C2$'s incoming tree, it is scheduled before the operations in $C2$. Then, the earliest schedulable time for $C2$ is computed as $e\_time(C2) = 4$, and the operations of $C2$ are scheduled.

Since no more unscheduled recurrence cycles are left, the scheduler handles the operations 6 and 12. Their schedule is determined by their (already scheduled) neighbors.

4.3 Scheduling of Mutually Dependent Recurrence Cycles

If two or more recurrence cycles have data dependencies between them, the scheduler gets stuck because it cannot compute a scheduling order for the clustered nodes because those nodes still form a cycle in the DFG. To handle such cases, we propose the following heuristics: First, the scheduler picks the clustered node C_{max} whose corresponding recurrence cycle has the longest recurrence delay among all mutually dependent nodes. Then, it schedules that node as described in the previous section. The schedule of C_{max} is not complete because some of its producers are not scheduled yet, hence additional steps are added to the scheduling process for dependent nodes. First, we define two sets: one contains the mutually dependent clustered nodes (u_p), and the other one contains the clustered node that was scheduled without all of its producers being placed beforehand (t_p). u_p is initialized with the unscheduled clustered producers of C_{max}, and t_p is initialized with C_{max}. Next, the clustered node C whose corresponding recurrence cycle has the maximum recurrence delay (and thus has to satisfy the tightest time constraints) among u_p is selected and scheduled. If the scheduling succeeds, remove C from u_p. Then, repeat the scheduling process for the nodes in t_p. This step is necessary to schedule operations on the path from the just scheduled node C to nodes in t_p for which that scheduler had ignored nodes in the incoming tree. Finally, the scheduler adds the unscheduled clustered producer of C to u_p and adds C to t_p. This process is repeated until the set u_p is empty.

If scheduling fails during this process, the scheduler retries scheduling by placing C_{max} to another slot. If there are no more slots available for C_{max}, the scheduler concludes that it is impossible to find a valid schedule for the given II.

Algorithm 1 shows the pseudo code for this process. The pseudo code starts after C_{max} has been scheduled.

The proposed heuristics successfully resolves deadlocks that occur due to mutually data dependent recurrence cycles. If one of the recurrence cycles fails to be scheduled, the whole process is restarted. This increases the compilation time, but prevents scheduling failures that arise due to redundant time constraints of operations that were scheduled before the recurrence cycles themselves.

Algorithm 1 Algorithm for handling deadlock situations

$u_p \longleftarrow$ *unscheduled clustered producer of* C_{max};
$t_p \longleftarrow \{C_{max}\}$;
while $u_p \neq \emptyset$ **do**
  $C \longleftarrow$ cycle with max. recurrence delay from u_p
  $PlaceIncomingTree(C)$;
  $PlaceLoop(C)$;
  $u_p \longleftarrow u_p - \{C\}$;
  for $C' \in t_p$ **do**
    $PlaceIncomingTree(C')$;
    $PlaceLoop(C')$;
  end for
  for $P \in$ *unplaced clustered producer* C **do**
    $u_p \longleftarrow u_p + \{P\}$;
  end for
  $t_p \longleftarrow t_p + \{C\}$;
end while

5. Architecture Modification

5.1 Limitations of Existing Architectures

A CGRA is a loop accelerator that operates in data-flow mode, hence a mechanism to get live-in operands into and live-out operands out of the CGRA is required. Often, the memory bandwidth is the bottleneck of a loop, so placing live values in a register file is usually preferable but also has a drawback: the number of FUs directly connected to the RF containing the live values has to be limited since both a long wire delay between the RF and the FU as well as RFs with many read/write ports hinder high clock frequencies. This limitation is a serious obstacle because it forces the scheduler to place operations that access live values on the few FUs connected to the RF.

Even though the proposed algorithm improves the quality of the schedules, it does not perform very well for some specific data-flow patterns. We have identified two significant data-flow patterns that the scheduler has difficulties to deal with.

The first one is a recurrence cycle that accesses a live-in/out value. We have shown earlier that innate time constraints hampers the scheduling of recurrence cycles. For this reason, most modulo scheduling algorithms process operations that are part of a recurrence cycle prior to the other operations. Even though the algorithm proposed in this work schedules recurrence cycles as soon as all of their producers have been scheduled, it might be relatively late in the scheduling process, especially for recurrence cycles whose incoming tree contains many operations. This poses a problem to the scheduler for recurrence cycles that access live-in/out values, i.e. operands that are both live-in and live-out of the loop. Live-in/out values must be accessed through a special RF, and the number of resources that have direct access to that RF is limited. Operations that read or write live-in/out values must be placed on those resources. This not only imposes a time, but also a resource constraint on those operations of the recurrence loop. Since 95% of all recurrence cycles access live-in/out values this limitation cannot be ignored.

The second data-flow pattern that is difficult to find an efficient schedule is related to the routing of live-in values. Live-in operands of a loop have two noticeable characteristics. First, they have a higher fan-out than normal operands. Tab. 1 shows that live-in operands have a fan-out more than twice as large compared to normal operands. The second characteristic is that live-in values possess long live ranges. This is related to the high fan-out because a high number of consumer operations are likely to prolong the live range of the producer. Another reason for the long life ranges is that memory operations usually receive the base address operand through a live-in value. Live-in values are generally placed at the top of the DFG and (especially) store operations reside at the bottom of the DFG, hence the long live range.

Application	live-ins	other operations
3D rendering	1.88	1.29
AAC decoder	2.91	1.31
AMR-WB+ decoder	3.02	1.19
eAAC+ decoder	2.90	1.33
H.264 decoder	5.86	1.40
mp3 decoder	3.62	1.32
MPEG surround decoder	2.51	1.36
Average	3.24	1.53

Table 1. Average number of fan-outs for live-in and other operations.

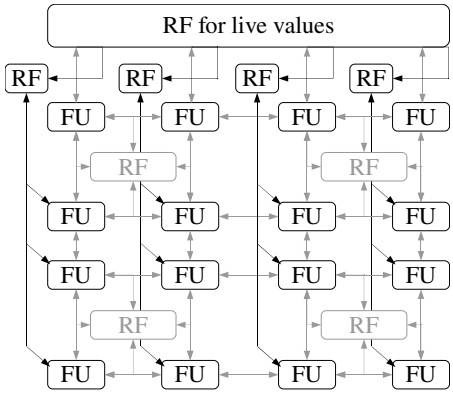

Figure 7. CGRA architecture with dedicated register files for live values. Additional connections to the dedicated register files are accentuated.

5.2 Proposed architecture

To overcome the limitations described in previous section, we propose to add a dedicated register file to each read port of the RF that contains the live values. The dedicated RF stores the values read from the read port it is connected to. The dedicated RF itself is connected to all FUs that are located in the same column as the FU that is directly connected to said read port. Note that accessing a live value through the dedicated RFs takes one extra cycle. Fig. 7 shows a 4x4 architecture with the proposed dedicated register files.

The dedicated RFs have two positive effects: first, the resource constraint imposed on scheduling recurrence cycles that access live-in/out values is relieved since more FUs than just the ones directly connected to the RF containing the live values can now read/write live values. This restriction could be alleviated with a shared bus that connects all FUs to the RF containing the live values. However, the physically long wire to FUs located far away from the RF would increase the critical path delay and thus hamper high clock frequencies. Dedicated RFs do not suffer from that problem because it takes one additional cycle to access a live value through a dedicated RF.

			Total II			Performance Loss (%)		
Application	Number of loops	MII	DRESC	HBEMS	RAMS	DRESC	HBEMS	RAMS
3D rendering	80	290	305	333	315	5.1	14.8	8.6
AAC decoder	35	123	141	173	137	14.6	40.6	11.4
AMR-WB+ decoder	44	230	233	247	235	1.3	7.4	2.2
eAAC+ decoder	47	204	217	248	214	6.4	21.6	4.9
H.264 decoder	61	387	470	459	456	21.4	18.6	17.8
mp3 decoder	8	44	45	54	45	2.3	22.7	2.3
MPEG surround decoder	115	522	554	582	557	6.1	11.5	6.7
Overall	390	1800	1965	2099	1959	9.2	16.6	8.8

Table 2. Comparison of the scheduling quality.

Second, the dedicated RFs increase the output bandwidth for live-in values because the same value can be retained for several cycles. Moreover, it also reduces the number of resources used for routing live-in values since all FUs now have indirect access to the live-in values. Our analysis shows that without dedicated RFs most of the routing resources, especially local registers, were occupied by live values routes. The additional dedicated RFs, however, cover most of the live value routing, and the remaining routing resources can be used for the routing of intermediate results.

The experimental results in Section 6 show that, keeping the total number of registers constant, the the proposed scheduling method computes better schedules using dedicated register files than one central RF containing the live-in/out values.

6. Experimental Results

6.1 Experimental Setup

We evaluate the proposed scheduling algorithm on a target architecture consisting of 16 functional units arranged on a 4x4 grid. Four FUs support memory access operations, and six FUs contain a multiplier. Each FU is directly connected to its maximal eight neighboring FUs and has exclusive access to a local rotating RF with eight entries. The local RFs are also connected with each of its neighboring local RF. A central register file with 64 non-rotating entries is reserved for live-in/out operands.

The loops to be mapped onto the CGRA are taken from six real-life applications, a 3-D renderer and five multimedia decoders: AAC, AMR-WB+, eAAC+, H.264, MP3, and MPEG surround. We have extracted a total of 390 loops from the applications with the number of operations per loop ranging from 47 to 393.

We compare the proposed recurrence cycle aware modulo scheduler (RAMS) to two different modulo schedulers: DRESC [11], a modulo scheduler based on simulated annealing, and a height-based EMS (HBEMS) [14] that schedules loops in an edge-centric way with a simple height-based scheduling priority scheme. All experiments were run on an Intel DualCore Xeon with a clock frequency of 2.66 GHz and 2 GB of memory. The compilation time was measured using only one core.

6.2 Comparison of the Scheduling Quality

Tab. 2 shows the scheduling quality of the different CGRA schedulers by comparing the initiation interval (II) of the final schedules. The second column displays the number of scheduled loops, and column three contains the accumulated minimal II (MII) of all loops. Columns four to six show the accumulated II of all loops for DRESC, HBEMS, and RAMS, respectively. Columns seven to nine, finally, show the percentual increase of each scheduler relative to the optimal (i.e., the accumulated MII).

The results show that the scheduling quality of the proposed scheduler matches that of the DRESC framework which sacrifices compilation speed in order to obtain a high scheduling quality. Comparing the proposed RAMS to HBEMS shows that the proposed algorithm achieves a significantly better scheduling quality even though both schedulers are based on the fast edge-centric scheduling technique.

Breaking down the results per benchmark shows that the AAC and H.264 decoders perform worse than average. An analysis of the benchmarks reveals three types of loops for which it is difficult to compute a schedule close to MII for any CGRA scheduler: the first type are loops with an MII of 1. For such loops, there is only one time slot available for each resource, hence there is no room for compromise when two different operations require a scarce resource (e.g., an FU that support both memory operations and multiplication). The AAC.1 decoder contains a large number of loops with $MII = 1$ which is why the overall scheduling quality is relatively low. The second type are complex loops consisting of a large number of operations. Such loops naturally also have a large number of edges. Since the MII is calculated considering the available computing resources only but ignoring the routing of operands, the routing pressure for a given MII is very high for such loops. In fact, it may even be impossible to schedule the loop at MII because there are simply not enough routing resources available. The H.264 decoder contains a loop with 393 operations and an MII of 13, however, a schedule for that loop can only be found at $II = 17$. The third type of loops that are difficult to find a schedule close to MII are loops whose DFGs are narrow, but deep. Such loops often have many data-flow edges that connect operations relatively far away from each other, that is, the scheduler has to route the operand over a long distance. Several loops in our 3-D renderer fall into this category.

6.3 Comparison of the Compilation Time

Tab. 3 compares the three schedulers in terms of compilation time. The first two columns display the number of loops per benchmark. Columns three to five contain the absolute compilation time as obtained on our test system. The sixth column shows the speed-up of RAMS compared to DRESC, and the seventh column compares the speed of the HBEMS framework in relation to RAMS.

		Compilation time(sec)			Speed Up	
Application	Number of loops	DRESC	HBEMS	RAMS	DRESC	HBEMS
3D rendering	80	43837.71	223.38	159.21	275.35x	1.40x
AAC decoder	35	34789.18	15.49	29.15	1193.45x	0.53x
AMR-WB+ decoder	44	20214.31	31.34	79.89	253.02x	0.39x
eAAC+ decoder	47	25290.63	23.84	63.84	396.16x	0.37x
H.264 decoder	61	56804.32	458.63	676.65	83.95x	0.67x
mp3 decoder	8	17313.74	12.56	30.34	570.66x	0.41x
MPEG surround decoder	115	65460.45	51.99	509.59	147.80x	0.12x
Overall	390	263710.34	817.23	1548.67	170.28x	0.53x

Table 3. Comparison of the compilation time.

The results show that both edge-centric schedulers, HBEMS and RAMS, outperform DRESC by two orders of magnitude. While the proposed scheduling methods is about two times slower than HBEMS, the absolute numbers show that it is still sufficiently fast to be used in a production environment. Moreover, the superior scheduling quality of RAMS over HBEMS compensates for this slowdown.

The reason for the slowdown of the proposed scheme compared to HBEMS stems from the different handling of recurrence edges in the DFG. In the HBEMS framework recurrence edges are treated equally to normal edges. In the proposed scheme, however, recurrence cycles in the DFG are scheduled as a single unit. This means that if any one of the recurrence edges comprised in a single recurrence cycle fails to be scheduled, the schedule of all other recurrence edges is also rolled back and rescheduled. The effect on the compilation time is more noticeable for loops with mutually-dependent recurrence cycles resulting in a deadlock. If only one of both cycles fails to schedule, both recurrence cycles are rescheduled. The MPEG surround decoder contains a significant number of loops whose DFGs form mutually-dependent recurrence cycles, hence the difference in compilation time is most pregnant for this benchmark.

6.4 The Effect of Dedicated Register Files

To analyze the effect of the dedicated RFs as proposed in Section 5 we compare the given architecture to one without dedicated RFs where only the FUs in the first row can access the RF containing the live values. To keep the total number of registers constant, we increase the number of entries in the local register files from 8 to 12.

Tab. 4 compares the accumulated II of all schedules of the architecture without dedicated RFs to that with dedicated RFs. On average, the proposed architecture with the dedicated RFs outperforms the unmodified architecture by 13% even though both architectures contain the same number of registers. In addition, comparing Tab. 4 to Tab. 2 reveals that the two applications with a high live-in fan-out (H.264 and MP3) profit most from the dedicated RFs.

7. Conclusion

In this paper, we describe a new recurrence cycle-aware scheduling technique for CGRA processors. Existing modulo schedulers that produce good-quality schedules suffer from excessive compilation times (such as the ones based on simulated annealing), others run reasonably fast, but do produce suboptimal schedules. Our approach overcomes these problems and produces schedules of good quality with short compilation times. The method is based on an edge-centric modulo scheduler. Instead of applying simple height-ordering to determine the scheduling order, nodes that form a recurrence cycle in the data-flow graph are first grouped and subsequently treated as one clustered node. The scheduler then focuses on scheduling the clustered nodes first. Clustered nodes which require lots of computations before their loop head can be placed are handled first.

We present deadlock-resolving heuristics for cases when any two (or more) recurrence cycles depend on each other. In such cases, the scheduler gives the recurrence cycle with the longest recurrence delay the highest priority.

We have identified and analyzed two types of loops that are difficult to schedule for modulo schedulers. We propose to alleviate the problem by adding dedicated register files for live values to provide access to those values to more functional units. Our experiments show that the scheduler computes, on average, 13% more efficient schedules for the hardware with dedicated register files while keeping the total number of registers constant.

We have implemented the proposed method into our in-house CGRA chip and compiler solution which is used for several next-generation multimedia devices. The experimental results show that the proposed technique achieves better quality schedules than the technique based on simulated annealing at a 170-fold speed increase.

References

[1] 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 *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 363–368, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[2] N. Bansal, S. Gupta, N. Dutt, A. Nicolau, and R. Gupta. Interconnect-aware mapping of applications to coarse-grain reconfigurable architectures. *Lecture Notes in Computer Science*, 3203:891–899, 2004.

[3] J. C. Dehnert and R. A. Towle. Compiling for the cydra 5. *The Journal of Supercomputing*, 7(1-2):181–227, 1993.

[4] C. Ebeling, D. C. Cronquist, P. Franklin, J. Secosky, and S. G. Berg. Mapping applications to the rapid configurable architecture. In *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, page 106, Washington, DC, USA, 1997. IEEE Computer Society.

	Total II		.
Application	w.o. Dedicated RF	w. Dedicated RF	Performance Improvement (%)
3D rendering	347	315	9.3
AAC decoder	160	137	14.4
AMR-WB+ decoder	245	235	4.1
eAAC+ decoder	248	214	13.7
H.264 decoder	546	456	16.5
mp3 decoder	62	45	27.3
MPEG surround decoder	651	557	14.4
Overall	2259	1959	13.3

Table 4. Effect of the dedicated register files on the II

[5] S. Goldstein, H. Schmit, M. Moe, M. Bidu, S. Cadambi, R. Taylor, and R. Laufer. Piperench: a coprocessor for streaming multimedia acceleration. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 28–39, Washington, DC, USA, 1999. IEEE Computer Society.

[6] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *DATE '01: Proceedings of the conference on Design, Automation and Test in Europe*, pages 642–649, Washington, DC, USA, 2001. IEEE Computer Society.

[7] A. Hatanaka and N. Bagherzadeh. A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In *Parallel and Distributed Processing Symposium*, pages 1–8, Washington, DC, USA, 2007. IEEE Computer Society.

[8] J.-e. Lee, K. Choi, and N. D. Dutt. Compilation approach for coarse-grained reconfigurable architectures. *IEEE Design & Test of Computers*, 20(1):26–33, 2003.

[9] J. Llosa. Swing modulo scheduling: A lifetime-sensitive approach. In *PACT '96: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, page 80, Washington, DC, USA, 1996. IEEE Computer Society.

[10] G. Lu, H. Singh, M.-H. Lee, N. Bagherzadeh, F. J. Kurdahi, and E. M. C. Filho. The morphosys parallel reconfigurable system. In *Euro-Par '99: Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, pages 727–734, London, UK, 1999. Springer-Verlag.

[11] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10296, Washington, DC, USA, 2003. IEEE Computer Society.

[12] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins. Design methodology for a tightly coupled vliw/reconfigurable matrix architecture: A case study. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 21224, Washington, DC, USA, 2004. IEEE Computer Society.

[13] H. Park, K. Fan, M. Kudlur, and S. Mahlke. 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*, pages 136–146, New York, NY, USA, 2006. ACM.

[14] 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 *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 166–176, New York, NY, USA, 2008. ACM.

[15] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74, New York, NY, USA, 1994. ACM.

[16] 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 *ASP-DAC '08: Proceedings of the 2008 conference on Asia and South Pacific design automation*, pages 776–782, Los Alamitos, CA, USA, 2008. IEEE Computer Society Press.