Edge-centric Modulo Scheduling for Coarse-Grained Reconfigurable Architectures

Hyunchul Park, Kevin Fan, and Scott Mahlke
Advanced Computer Architecture Laboratory,
University of Michigan
Ann Arbor, MI, USA
{parkhc, fank, mahlke}@umich.edu

Taewook Oh, Heeseok Kim, and Hong-seok Kim
Samsung Advanced Institute of Technology
Kiheung, Republic of Korea
{taewook.oh, heeseok.kim, hong-seok.kim}@samsung.com

ABSTRACT

Coarse-grained reconfigurable architectures (CGRAs) present an appealing hardware platform by providing the potential for high computation throughput, scalability, low cost, and energy efficiency. CGRAs consist of an array of function units and register files often organized as a two dimensional grid. The most difficult challenge in deploying CGRAs is compiler scheduling technology that can efficiently map software implementations of compute intensive loops onto the array. Traditional schedulers focus on the placement of operations in time and space. With CGRAs, the challenge of placement is compounded by the need to explicitly route operands from producers to consumers. To systematically attack this problem, we take an edge-centric approach to modulo scheduling that focuses on the routing problem as its primary objective. With edge-centric modulo scheduling (EMS), placement is a by-product of the routing process, and the schedule is developed by routing each edge in the dataflow graph. Routing cost metrics provide the scheduler with a global perspective to guide selection. Experiments on a wide variety of compute-intensive loops from the multimedia domain show that EMS improves throughput by 25% over traditional iterative modulo scheduling, and achieves 98% of the throughput of simulated annealing techniques at a fraction of the compilation time.

Categories and Subject Descriptors

D.3.4 [**Processors**]: [Code Generation and Compilers]; C.3 [**Special-Purpose and Application-Based Systems**]: [Real-time and Embedded Systems]

General Terms

Algorithms, Experimentation, Performance

Keywords

Coarse-grained Reconfigurable Architecture, Operand Routing, Programmable Accelerator, Software Pipelining

1. INTRODUCTION

The embedded computing systems that power today's portable devices demand high performance and energy efficiency. Traditionally, application specific hardware in the form of ASICs has been used on the compute-intensive kernels to meet these demands. However, increasing convergence of different functionalities, such as voice/data communication, high definition video, and digital photography on a single device, combined with high non-recurring costs involved in designing ASICs, have pushed designers towards programmable solutions. Coarse-grained reconfigurable architectures (CGRAs) are becoming attractive alternatives because they offer large raw computation capabilities with low cost/energy implementations. Example CGRA systems that target wireless signal processing and multimedia are ADRES [15], MorphoSys [13], and Silicon Hive [19]. Tiled architectures, such as Raw, are closely related to CGRAs [22].

CGRAs generally consist of an array of a large number of function units (FUs) interconnected by a mesh style network. Register files are distributed throughout the CGRA to hold temporary values and are accessible only by a small subset of FUs. The FUs can execute common word-level operations, including addition, subtraction, and multiplication. In contrast to FPGAs, CGRAs sacrifice gate-level reconfigurability to increase hardware efficiency. As a result, they have short reconfiguration times, low delay characteristics, and low power consumption.

An effective compiler is essential for exploiting the abundance of computing resources available on a CGRA. However, sparse connectivity and distributed register files present difficult challenges to the scheduling phase of a compiler. Traditional schedulers that just assign an FU and time slot to each operation are insufficient because they do not take routing into consideration. Scalar operand values must be explicitly routed between producing and consuming operations. Further, dedicated routing resources are not provided. Rather, an FU can serve either as a compute resource or as a routing resource at a given time. A compiler scheduler must manage the computation and flow of operands across the array to effectively map applications onto CGRAs.

To efficiently make use of the CGRA resources, modulo scheduling (or other software pipelining variations) of loops is generally used [20]. This provides the opportunity to exploit both loop-level and instruction-level parallelism to efficiently make use of the CGRA resources. To deal with the complex topology and routing challenges, the DRESC (Dynamically Reconfigurable Embedded System Compiler) proposes a modulo scheduling algorithm based on simulated annealing [14]. It begins with a random placement of operations on the FUs, which may not be a valid modulo schedule. Operations are then moved between FUs until a valid schedule is achieved. The strength of simulated annealing is its ability to deal with both sparse connectivity and complex resource usage that are common in a CGRA. DRESC consistently achieves the leading

PACT'08, October 25-29, 2008, Toronto, Ontario, Canada.

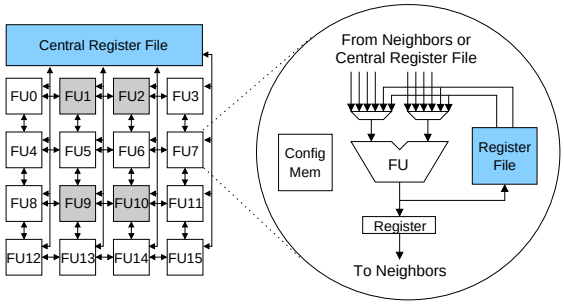

Figure 1: Example CGRA design.

performance results over other methods on a variety of CGRAs. However, the random movement of operations in the simulated annealing technique can result in a long convergence time for loops with modest numbers of operations. Also, the algorithm is ad-hoc in the sense that no information about the structure of the loop's dataflow graph is utilized in making scheduling decisions.

For this work, our goal is to develop a more systematic approach where compilation time is a first-class constraint. We initially chose to adapt iterative modulo scheduling to CGRAs because it both produces efficient results and offers short compilation times even for large loops [20]. The central changes were adapting the scheduler to understand the decentralized resources of a CGRA as well as performing routing of operands between producing and consuming operations. While this approach was successful at creating correct schedules, loop throughput was reduced by 10-50% in comparison to the simulated annealing method. An analysis of the resultant loops showed that *node-centric modulo scheduling* is a poor match for CGRAs. Traditional schedulers are node-centric in that the focus is assigning operations (nodes) to FUs. The straightforward adaptation of this approach is operation assignment followed by operand routing to determine if the assignment is feasible. However, even with large numbers of free FUs, the scheduler inevitably fails due to the inability to route an operand. Further, backtracking is ineffective due to the complex interrelations between scheduler decisions.

The key insight from this experience was that a CGRA scheduler must consider routing efficiency as the primary objective. Selecting intelligent paths from producing to consuming FUs that do not block other operand paths is essential to achieving higher throughput schedules. Further, operation assignment can be viewed as a by-product of a successful route, thus no successive placement step is required. In essence, by getting an operand between two points, the necessary operations can be performed along the way for free. We refer to this technique as *edge-centric modulo scheduling*, or EMS. This paper presents the design, implementation, and evaluation of the EMS algorithm.

2. BACKGROUND AND MOTIVATION

2.1 Architecture Overview

A CGRA consists of an array of compute nodes, each of which executes word-level operations, communicating through an interconnection network. In general, CGRA designs can be described by four characteristics: size, node functionality, network configuration, and register file sharing. The *size* refers to the number of nodes; commonly this can vary from 4 nodes arranged in a row up to 64 nodes arranged in an 8×8 grid. The *functionality* of each node can vary from a single FU (e.g. adder or subtracter), to an ALU, to a full-blown processor. In addition, the functionality of nodes may be homogeneous or heterogeneous. For example, only a subset of nodes may access data memory.

There are a large number of potential *network configurations*, such as connections between each node and its four (or eight diagonal) nearest neighbors, buses connecting each node to (possibly to a subset of) other nodes in the same row or column, hierarchical connection schemes, and so on. Finally, the degree of *register file sharing* ranges from small, individual register files at each node, to multiple register files each shared by a small number of nodes, to a single central register file accessible by some or all nodes.

Figure 1 shows an example CGRA design that contains 16 nodes arranged in a 4×4 mesh; each node can communicate with its four nearest neighbors. In addition, column buses connect each node to a central register file. Each node consists of an FU that can read inputs from neighbors or the central register file and write to a single output register; a small, dedicated register file; and a configuration memory to supply control signals to the MUXes, FU, and register file. Certain operations, such as loads and stores, can only be executed on a subset of FUs (shaded). Note that a node can either perform a computation or route data each cycle, but not both, as routing is accomplished by passing data through the FU (a MOVE operation).

2.2 Modulo Scheduling Challenges

Modulo scheduling is a software pipelining technique that exposes parallelism by overlapping successive iterations of a loop [20]. The goal is to find a valid schedule such that the interval between successive iterations (initiation interval, or II) is minimized. The II-cycle code region that achieves this maximal overlap is called the kernel. When the number of iterations is large, the performance of the loop is determined by the II to a first order; thus, it is more important to minimize the II than to minimize schedule length. Initially, the scheduler chooses the target II to be the maximum of the resource-constrained lower bound (ResMII) and the recurrence-constrained lower bound (RecMII). If a valid modulo schedule cannot be found, the target II is incremented and scheduling is attempted again.

Scheduling for CGRAs is quite different from scheduling for general VLIW architectures due to the different hardware characteristics. Factors that complicate CGRA scheduling include:

Explicit routing. In a VLIW architecture, routing from producer to consumer is implicitly guaranteed by storing intermediate values in a multi-ported, centralized register file. However, in a CGRA, interconnect is much more sparse and values must be explicitly routed using FUs, local register files, and mesh connections.

Intelligent routing. FUs are used for both computation and routing; thus, scheduling can easily fail if poor routing choices are made. Furthermore, the scheduler must not only generate a valid schedule, but also minimize the routing resources used so that more FUs are available for computation.

Heterogeneous nodes. All nodes can perform addition and logical operations, but "expensive" operations such as multiplies, loads, and stores may only be supported by a subset of nodes. In such an architecture, it is important to avoid scheduling inexpensive operations on expensive nodes, because this limits the scheduling flexibility of the expensive operations.

Modulo constraint. Resources are used in a periodic fashion, since the loop kernel repeats every II cycles. Thus, unlike in acyclic scheduling, it is not possible to guarantee routability by extending the schedule, and scheduling can easily fail due to the previously scheduled operations.

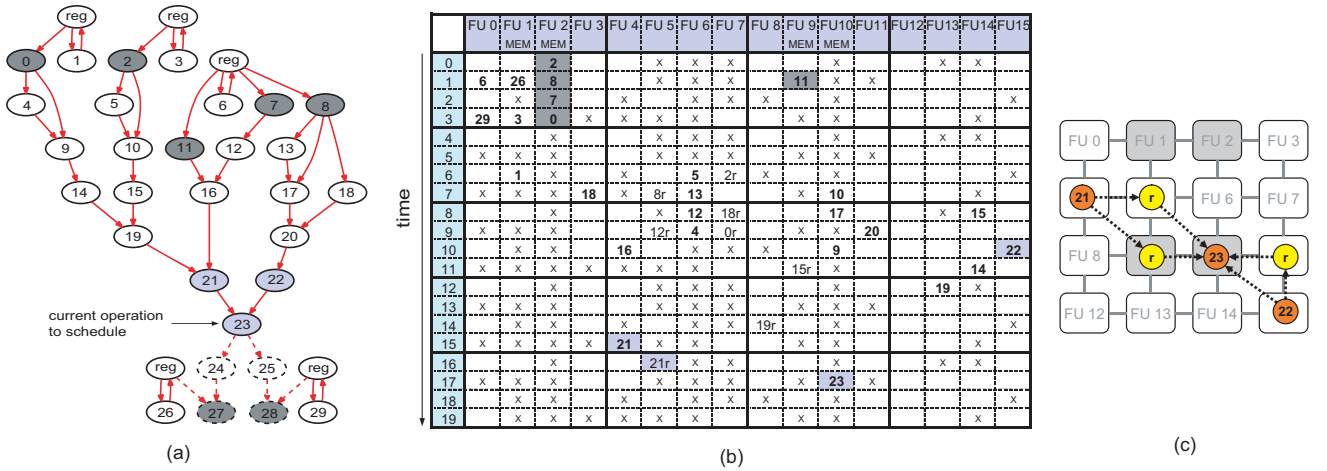

Figure 2: Example to illustrate the challenges of CGRA scheduling: (a) the dataflow graph for the fsed application, (b) the reservation table for a partial schedule on a 4x4 array, (c) possible routings from 23's producers. In (a) and (b), dark grey shading indicates memory operations and light grey shading is used to highlight the current operation being scheduled (node 23) and its immediate predecessors. Bold numbers indicate computation operations, other numbers followed by 'r' (e.g. '8r') indicate routing slots for corresponding computation operations. 'reg' nodes indicate live-in values stored in the central RF.

To illustrate the complexities of CGRA modulo scheduling, Figure 2(a) shows the dataflow graph (DFG) for the dominant loop from one of our benchmark applications, `fsed`, an image halftoning algorithm. Memory operations are shaded dark grey. The DFG is being scheduled onto a 4×4 CGRA, similar to the one shown in Figure 1, with II=4. The partial schedule is shown in Figure 2(b). schedule is shown. Bold numbers are computation operations; other numbers followed by 'r' (e.g. '8r') are routing operations for the corresponding computation operations; and, Xs represent slots that are occupied due to the modulo constraint. 'reg' nodes indicate live-in values that are stored in the central RF. All operations above operation 23 (light grey) in the DFG have been scheduled at this point.

There are several points to observe. First, only FUs 1, 2, 9, and 10 support memory operations, thus all of the memory operations must be scheduled on those FUs. Next, observe how values are routed to operation 23, which is considered for execution on FU 10 at time 17, and has two producers: 21 and 22. Figure 2(c) shows the possible routes of the operands from two producers. One possible way to route the operand from 21 to 23 is through FU 9. The operand is first routed diagonally from FU 4 to FU 9 via a shared register fie, then it is routed to the neighboring FU 10 via the mesh connection. However, taking this option leaves only two memory slots for the unscheduled memory operations (27 and 28). Therefore, the operand of 21 is routed through FU 5 rather than through FU 9. Similarly, the operand of 22 is routed directly from FU 15 to FU 10 rather than through FU 11. The value is stored in a rotating register file for 6 cycles and is read out by 23 at time 17. The challenge here is how to guarantee the availability of storage in the register file. The available storage must be carefully considered during scheduling as simply pushing register allocation to after scheduling can result in costly spilling and may require complete rescheduling of the loop. It can be seen that routing is complex, and various resources including FUs, registers, register file ports, and connection links must be modeled by the compiler to properly orchestrate the flow of values from producers to consumers. Further, this routing adds latency to the schedule: operation 23 has an earliest start time of 11, but is actually scheduled at time 17.

3. CORE CONCEPTS

Prior to describing the EMS algorithm, we describe several of the important concepts along with their rationale. These concepts are described in isolation (and hence will appear disconnected), but they are tied together in Section 4.

3.1 Integrated Placement and Routing

CGRA scheduling can be broken down into two tasks: placement of operations into computation slots (FU and time) and routing of operands. Previous techniques ([14], [18]) address the scheduling problem in a node-centric manner, meaning that the scheduler places operations first and then does the routing. When an operation is scheduled, it is placed in a slot where it can execute, and operands from other producers or consumers are then routed to the scheduled slot. However, scheduling failures usually occur during the routing phase because of the limited connectivity between resources. In this work, we propose an edge-centric approach where the scheduler primarily focuses on routing, and placement occurs during the routing process.

Node-centric Approach. Node-centric approaches place operations in a way that minimizes a heuristic routing cost. The routing cost consists of various metrics that determine the quality of placement (e.g., the number of resources used for routing) [18]. The scheduler visits candidate slots one by one until it finds a solution. The operation is placed in each candidate slot, and edges to the placed producers and consumers are routed. Figure 3(b) shows how an optimal placement is found with this approach. A DFG containing two producers P1 and P2 and a shared consumer C is mapped onto the hypothetical 1×5 CGRA in Figure 3(a). For illustration purposes, we assume no register file in this architecture. P1 and P2 are already placed and the scheduler places the consumer C by visiting all the empty slots as shown in Figure 3. The slots with dotted circles are failed attempts where the scheduler could not route values from P1 or P2 due to resource conflicts. After visiting those slots, the scheduler successfully places C on FU 4 at time 4 (slots will be referred as (FU #, time) hereafter).

One can observe two inefficiencies with this approach. First, the scheduler makes unnecessary visits to empty slots (0,2), (0,3),

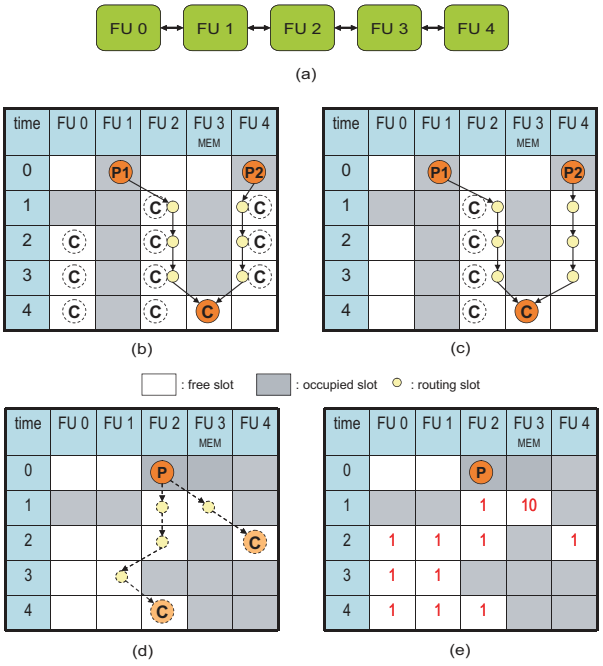

Figure 3: High level comparison of scheduling approaches: (a) 1x5 CGRA, (b) compile time example of node-centric, (c) compile time example of edge-centric, (d) performance example of node-centric, (e) performance example of edge-centric. Shaded boxes in the reservation tables indicate slots occupied by other operations.

and (0,4). This is because the scheduler places operations without routing information. The second inefficiency is that there are redundant routings made when the scheduler visits (2,1), (2,2), (2,3), (2,4), and (3,4). For example, when the scheduler visits slot (3,4), it already knows that there is a path P1→(2,1)→(2,2)→(2,3) since it was discovered when slot (2,3) was visited. These observations show that placement without routing information can lead to redundant routing calls, which increases compilation time. One can argue that a different visiting order can solve this problem (visiting slots in the same FU first). Even though this can work for this particular case, there is no general order that works for all the cases in the node-centric approach.

A node-centric approach can also lead to a poor solution because it does not consider routing information when placing an operation. Figure 3(d) shows a different example where P is already placed and the edge from P to C is about to be routed. Here, we assume that C can be placed in only two slots, (4,2) and (2,4). Note that slot (3,1) is the only remaining memory access slot, thus it is critical to avoid using this slot for routing if possible. Since the node-centric approach visits slot (4,2) before slot (2,4), it will simply choose the path to slot (4,2) in Figure 3(d), using the memory slot for routing. If any memory operation still needs to be scheduled, the II must be increased. Here, we are assuming that the node-centric approach visits slots in an increasing order of time. Although a different visiting order can give priority to slot (2,4) over slot (4,2), that particular order cannot be applied to general cases without routing information. In general, the node-centric approach needs to perform an exhaustive search of all the available slots to handle this problem.

Edge-centric Approach. In an edge-centric approach, the placement of an operation is integrated into the routing function, and the placement decision is deferred until routing information is discovered. When scheduling an operation, the scheduler does not place the operation up front. Instead, it picks an edge from the operation's previously-placed producers or consumers and starts routing the edge. The router will search for an empty slot that can execute the target operation, rather than routing towards a placed operation. Once a compatible slot is found, the target operation is placed in the slot and the scheduler continues routing edges to other producers or consumers.

Figure 3(c) shows the same example of Figure 3(b), but the consumer is scheduled using an edge-centric approach. The scheduler begins with the edge from P1 to C, instead of scheduling operation C directly. When an empty slot is encountered, the scheduler temporarily places the target operation and checks if there are other edges connected to the consumer; if so, it recursively routes those edges. For example, when the router visits slot (2,1) in Figure 3(c), it temporarily places C there and recursively calls the router function to route the edge from P2 to C. When it fails to route the edge from P2 to C, routing resumes from slot (2,1), not from P1, and a solution is eventually found at slot (3,4). So, slots (2,1), (2,2), (2,3), (2,4), and (3,4) are all visited in one routing call. Compared to 11 routing calls made for the edge from P1 to C in Figure 3(b), only one routing call is required to find the same solution in the edge-centric approach. The number of routing calls for the edge from P2 to C is same for both approaches (5 calls), as the router is only called for that edge if the edge from P1 to C is routed successfully.

The second benefit of an edge-centric approach lies in the aspect of solution quality. In the example in Figure 3(d), it is desirable not to use slot (3,1) for routing. The edge-centric approach avoids using the memory slot (3,1) for routing by assigning a higher cost to the slot as shown in Figure 3(e). Here, a cost of 10 was assigned to slot (3,1) and all the other slots were assigned a cost of 1. Then, the edge-centric approach will automatically find a path that avoids slot (3,1) by prioritizing the route path by cost. So, it successfully finds a path to slot (2,4) using the left path in Figure 3(d).

An edge-centric approach can perform faster and achieve a better result than a node-centric approach. However, it has a greedy nature in that it optimizes for a single edge at a time, and the solution can easily fall into local minima. There is no search mechanism in the scheduler at the operation level and every decision made in each step is final. We address this problem by employing intelligent routing cost metrics explained in the next section.

3.2 Routing Cost Metrics

The routing function is the basic building block of the edge-centric scheduler, and every scheduling task, including placement, occurs in the routing function. The final schedule is formed by calling the routing function for each edge in the DFG.

It is important to achieve a good mapping for each individual edge. The routing function needs to have a global perspective of the entire mapping since individual decisions affect the routing of other edges. The order in which the router visits each scheduling slot is determined by a *routing cost* associated with each slot. Thus, it is crucial to develop a good routing cost function.

There are two main objectives when routing a single edge:

- Minimize the number of routing resources used, to leave more slots available for routing other edges.
- Proactively avoid routing failure: avoid using resources that will block future routes, and reserve computation slots for expensive operations.

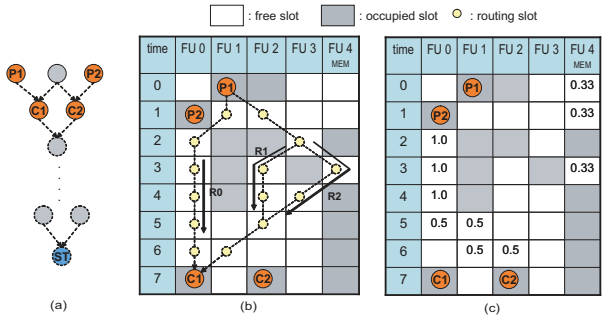

Figure 4: Routing cost example: (a) dataflow graph, (b) possible mappings, and (c) probabilistic cost.

3.2.1 Minimizing the Number of Routing Resources

Using the fewest routing resources is simple when considering a single edge. Each routing resource is assigned a statically-determined fixed cost, and the router will find a path that minimizes the total cost.

Typically, an operation is connected to multiple producers and consumers, so the router must consider the usage of routing resources when the other edges are routed as well. To address this issue, an *affinity cost* was proposed in previous work [18]. The affinity value for a pair of operations reflects their proximity in the DFG. In the edge-centric scheduler, each slot is assigned an affinity cost depending on how close it is to any already-placed operations that have high affinity with the target operation. This gives a preference for placing an operation near its producers and consumers, hence reducing the number of routing resources used.

3.2.2 Proactively Avoiding Routing Failure

Figure 4 gives an example of when naïve routing of an edge can lead to routing failures of other edges. The DFG on the left is mapped onto the example CGRA in Figure 3(a). The six operations at the top are being placed and the three at the bottom have not been placed yet. The operation ST at the bottom is a store operation; assume that only FU 4 can execute memory operations. When routing the edge from P1 to C1, there are three possible paths (R0, R1, and R2) as shown in Figure 4(b). All three paths use the same number of routing resources. However, there is a preferred choice when routing of other edges is considered. First, the path on the left (R0) should not be selected because it would block the only path between P2 and C2, causing a subsequent routing failure from P2 to C2. The path in the middle (R1) is preferred to the path on the right (R2) because occupying slot (4,3) leaves only two memory slots of FU4 for the ST operation. So, the scheduler will have fewer options when scheduling the ST, leading to a greater chance of routing failure in the future.

From the previous example, we can see that the scheduler needs to know the resources that are likely to be used by other edges in the future. To account for this, the scheduler associates an occupancy probability with each scheduling slot. The probabilities are calculated for two different types of operations: expensive operations and placed operations.

Expensive operations are defined as ones that only a subset of FUs can execute, such as memory and multiply operations. For each scheduling slot that can execute expensive operations, the probability is calculated by dividing the number of unscheduled expensive operations by the number of remaining slots that are compatible. When non-expensive operations are scheduled, the router prefers to avoid using slots that are capable of supporting expensive operations. For operations already placed in the scheduling space, the scheduler determines how many routing options there are for routing values to either producers or consumers.

For the placed operation P2 in Figure 4(c), probabilities are annotated in each reachable slot depending on the number of routing options. Empty slots in FU 4 are also annotated with a probability of 0.33 calculated by dividing the number of memory ops left by the number of available slots. These probabilities are accounted for when the routing cost is calculated for each slot, and the router will visit slots in the order of routing cost.

3.3 Stage Re-assignment

In modulo scheduling, better throughput (smaller II) is often achieved by scheduling some operations up front. A good example is operations on recurrence cycles. Since each iteration is executed every II cycles, all operations in the recurrence cycle must be scheduled within II cycles. For this reason, most modulo scheduling algorithms process operations on recurrence cycles prior to other operations.

When placing an operation in a recurrence cycle early in the scheduling process, it is likely that there are no producers or consumers placed already. In a conventional modulo scheduler, the scheduler utilizes ASAP/ALAP (as soon/late as possible) times calculated statically by looking at the longest paths between operations. In CGRA scheduling, the ASAP/ALAP time is not an accurate measure of the actual time slot because routing can take multiple cycles. If an operation is scheduled too early, the scheduler will fail to place its predecessors. If an operation is scheduled too late, there can be a waste of routing resources or increase in register pressure.

Accurate ASAP/ALAP times are not easily obtained in CGRA scheduling because they depend on routing latency which is not known a priori. Thus, we take an alternative approach: placed operations can be lowered or hoisted along the time axis by reassigning the stage. Since only stage count is changed, the resource occupancy status does not change. When an operation's stage is changed, operations connected to it in the scheduling space and routing between them must be moved as well. Since all the connected components are moved together, the stage reassignment is a local transformation and does not affect other operations.

An example of stage re-assignment is shown in Figure 5(a). Operations B and C form a recurrence cycle and are initially scheduled in stage 1 (times 2 and 3). Later, when operation A is being scheduled, the router is called for the edge from A to B. Since resources are repeatedly used every II cycles, FU 3's slot at time 6 is also occupied by operation B. Operations A and B are not connected by any placed edge, so B can be re-assigned to time 6 (in stage 3). Since operation C is connected to B by a placed edge, it is also re-assigned to time 7.

3.4 Edge Categorization

Modulo scheduling for the CGRA is a problem of allocating a fixed number of routing resources to the edges in the DFG. It is important to observe that not all edges are the same in terms of how important they are to the overall schedule. In EMS, edges in DFGs are categorized as described below, and different routing approaches are applied for each edge type.

Recurrence edges. It is crucial to schedule the edges in a recurrence cycle ahead of other operations, especially when the II is close to the length of the recurrence. These edges are thus scheduled with highest priority.

Simple edges and high-fanout edges. Simple edges are defined as the outgoing edge of an operation that has only one consumer.

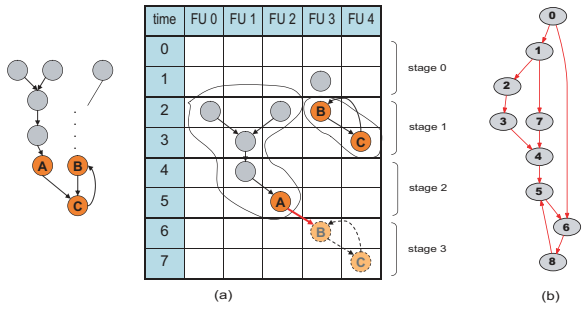

Figure 5: (a) Stage re-assignment example (II = 2) that re-assigns the recurrence cycle B-C from time 2-3 to time 6-7 after operation A is scheduled; (b) Example dataflow graph to illustrate non-critical edges.

When there are multiple consumers, the outgoing edges are called high-fanout edges. With the limited number of routing resources, edges routed earlier are likely to use less routing resources than edges routed later, since there is more flexibility when slots are not yet occupied. Therefore, the scheduler needs to intelligently decide which edges are routed first.

The edge-centric scheduler gives priority to simple edges over high-fanout edges for the following reason. When a simple edge is routed later and thus is not optimized very well, it will likely end up using more resources than required. Since there is no other consumer for the producer of the simple edge, those additional resources are just being wasted. However, additional resources in a high-fanout edge can actually be helpful when routing edges from the same producer to other consumers, since there are more resource slots that contain the producer's value.

An analysis on simulated annealing's result also shows this trend. Frequently, an operation that has multiple consumers is located far apart from its consumers on the time axis, while operations connected with simple edges are located close to each other. This observation motivates our priority calculation method using fanout clustering, described in the next section.

Non-critical edges. When there are multiple disjoint paths between a pair of nodes in the DFG, dependencies are generated between edges in different paths. An example is shown in Figure 5(b). Assume the recurrence cycle at the bottom (operations 5, 6, and 8) was scheduled first. When node 0 is scheduled, the scheduler sees that its consumer node 6 is already scheduled. However, the edge from 0 to 6 should not be routed yet because it is not on the critical path from 0 to 6. The scheduler should wait until all of the edges in the critical path are routed before routing the 0→6 edge. Therefore, a dependency is generated from the 0→6 edge to the critical path between 0 and 6. Similarly, dependencies are generated for edges on paths between nodes 1 and 4. In this case, edges 1→7 and 7→4 depend on the critical path between nodes 1 and 4. When an edge has a dependency on a pair of nodes, the routing of the edge is deferred until the edges on the critical path are scheduled.

4. IMPLEMENTATION

This section describes the implementation of EMS. The system flow is shown in Figure 8. First, the DFG of the target loop is converted into a reduced form by collapsing some nodes. The reduced DFG is then clustered by ignoring high-fanout edges and operations are prioritized based on the clustered result. Then, the operations are scheduled either by calling a placement function or calling a routing function depending on whether they have previously placed producers or consumers. After finding a legal schedule for the given II, the collapsed nodes are expanded first and configurations are generated for each component. If scheduling fails, the scheduler increases II and repeats scheduling.

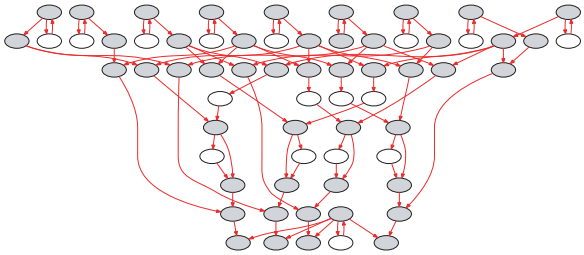

Figure 6: An example dataflow graph from H.264.

4.1 Prepass Steps

Generating the Reduced Dataflow Graph

First, the DFG is converted into a reduced form where certain nodes are collapsed into edges. An operation is collapsible if it is inexpensive (can execute on any FU in the array), and has only one producer and one consumer. When such a node is found, the scheduler removes it and draws an edge directly from its producer to its consumer. The new edge is annotated with the number of nodes that were collapsed. This simplifies the DFG, and also allows the router to treat a path of nodes as a single edge during routing, potentially leading to a better schedule for that path.

In the DFG in Figure 6, collapsible nodes are shown in white. When these nodes are collapsed into edges, a reduced DFG (RDFG) is generated as shown in Figure 7. In all, 17 out of 65 nodes were collapsed, resulting in a smaller scheduling problem. For the loops in the media applications evaluated in Section 5, 18% of nodes were collapsed on average.

Priority Calculation using Fanout Clustering

The scheduling priority of operations in the RDFG are calculated in such a way that simple edges get higher priority than high-fanout edges, as described in Section 3.4. First, the DFG is clustered by ignoring high-fanout edges. Each group of nodes connected by simple edges forms a cluster as shown in Figure 7. The scheduler processes clusters such that each cluster is scheduled as soon as all of its producers are placed. Within a cluster, producer operations are also scheduled before consumers. Basically, nodes are visited in a post-order traversal starting from the bottom.

For the target loop in Figure 7, the operations in recurrence cycles are scheduled up front. Then, the scheduling order of each cluster is determined. The scheduler will start with C8, which is one of the clusters at the bottom. A post-order traversal gives an order of C0, C3, C1, C4, C2, C7 and C8. The final order for clusters are C0, C3, C1, C4, C2, C7, C8, C5, C9, C6, C10, and C11. Within a cluster, operations are scheduled the same way.

4.2 Edge-centric Modulo Scheduler

Once priorities are calculated for all nodes in the RDFG, the nodes are scheduled. For each target operation, first the scheduler determines whether there are any placed producers or consumers. If not, the target operation is placed in a scheduling slot with minimum cost; this is the only time where the placement function is called. For an operation that has placed producers or consumers, the scheduler decides which edge to route first. The decision is made based on various factors such as schedule time and stage-changeability of producers or consumers, and how many routing options are available.

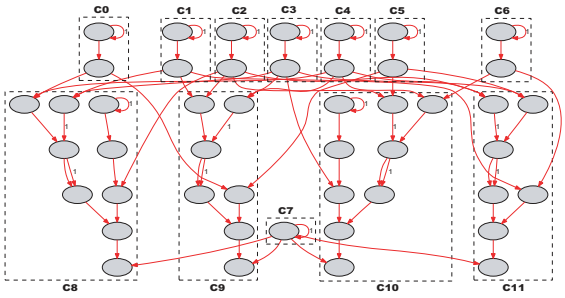

Figure 7: Example from Figure 6 after fanout clustering.

When an edge is selected, the router is called and it first decides the routing direction. Forward routing starts from the producer and finds a compatible slot for the consumer; backward routing does the opposite. When both producer and consumer are placed, both directions are possible, and the decision is made based on stage-changeability of the producer and consumer. Since only operations at the end of a route can have their stages re-assigned, the router will select a direction that starts from a fixed operation.

4.2.1 *Search Window Setup*

The router will visit neighboring scheduling slots starting from a slot where a source operation is placed. The scheduler needs to set up the time axis of the search window with care. A search window that is too small can result in failure to find a compatible slot, while there can be a waste of time if a window is too large. Even though ASAP/ALAP times are not an accurate measure of the time slots for operations to be placed, they can be a good lower/upper bound for routing. The search window is determined by ASAP/ALAP time of the target operation considering stage re-assignment. When routing an edge from a placed producer (*P*) to a non-placed consumer (*C*), ASAP time can be calculated by Equation 1. *p* denotes a placed predecessor of *C*. *d(x, y)* is the longest path delay between *x* and *y*. *up(x)* is the max number of stages *x* can be hoisted and *dn(x)* is the maximum number of stages *x* can be lowered. Similarly, ALAP time is calculated by Equation 2 where *s* denotes a placed successor of *C*.

$$ASAP(C) = MAX(time(p)+d(p,C)-(up(p)-dn(P))\times II) \quad (1)$$

$$ALAP(C) = MIN(time(s)-d(C,s)+(up(P)-dn(s))\times II) \quad (2)$$

4.2.2 *Routing Cost Calculation*

When scheduling an edge, a routing cost is calculated for each available slot. This cost is used by the router to determine the order in which to explore slots during routing. Routing cost has three primary components, described below.

Static cost. A fixed cost C_{static} is assigned to each slot so that the scheduler can minimize the number of routing resources used.

Affinity cost. As described in Section 3.2.1, affinity cost is calculated based on a slot's distance from placed producers. Equation 3 calculates the affinity between two operations *A* and *B*. Affinity is given to a pair of operations that have common consumers (direct or indirect use of the destination of *A* and *B*). Common consumers within *max_dist* in the DFG are considered for affinity calculation. *num_cons(A,B,d)* denotes the number of common consumers of *A* and *B* at the distance *d* in DFG.

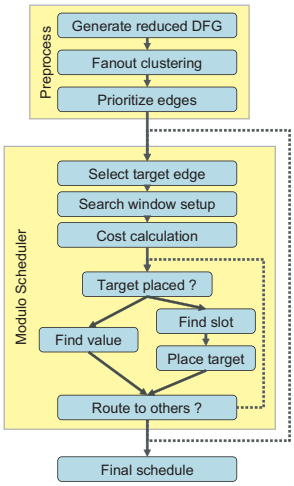

Figure 8: System flow for edge-centric modulo scheduling.

$$affinity(A,B) = \sum_{d=1}^{max\_dist} 2^{max\_dist-d} \times num\_cons(A,B,d) \quad (3)$$

The affinity cost C_{aff} is then calculated for each slot as follows, where $dist$ is the distance in hops from the current slot to the slot where the producer is placed. When there are multiple placed producers, C_{aff} is summed for all producers.

$$C_{aff} = \begin{cases} 0 & affinity(A,B) = 0 \\ \frac{dist}{affinity(A,B)} & affinity(A,B) > 0 \end{cases} \quad (4)$$

Probability cost. The router should take care not to block certain slots because they may be required for routing of future edges. Thus, a cost is assigned to each slot reflecting the probability that it will be required in the future. There are two cases: reserving expensive slots, and reserving slots to route results of previously placed nodes. The individual probabilities are calculated as described in Section 3.2.2. These probabilities must then be combined together, as a given slot may support multiple types of expensive operations and/or be used to route multiple placed nodes. Since the individual probabilities are correlated, getting the exact overall probability for a slot is difficult. An approximation is obtained by treating the probabilities independently. The following equation expresses the total probability P of a slot given n individual probabilities p_i:

$$P = \sum_{k=1}^{n} \left((-1)^{k-1} \sum_{\substack{I \subset \{1,\ldots,n\} \\ |I|=k}} \prod_{i \in I} p_i \right) \quad (5)$$

Total routing cost. The total routing cost C for a slot is obtained by combining the three costs above:

$$C = \begin{cases} C_{static} + w_{aff} \times C_{aff} + w_P \times P & P < 1 \\ \infty & P = 1 \end{cases} \quad (6)$$

The costs are combined with weighting factors w_{aff} and w_P. In addition, if $P = 1$, the slot will definitely be required in the future and cannot be used for routing the current edge; thus, routing cost is infinite.

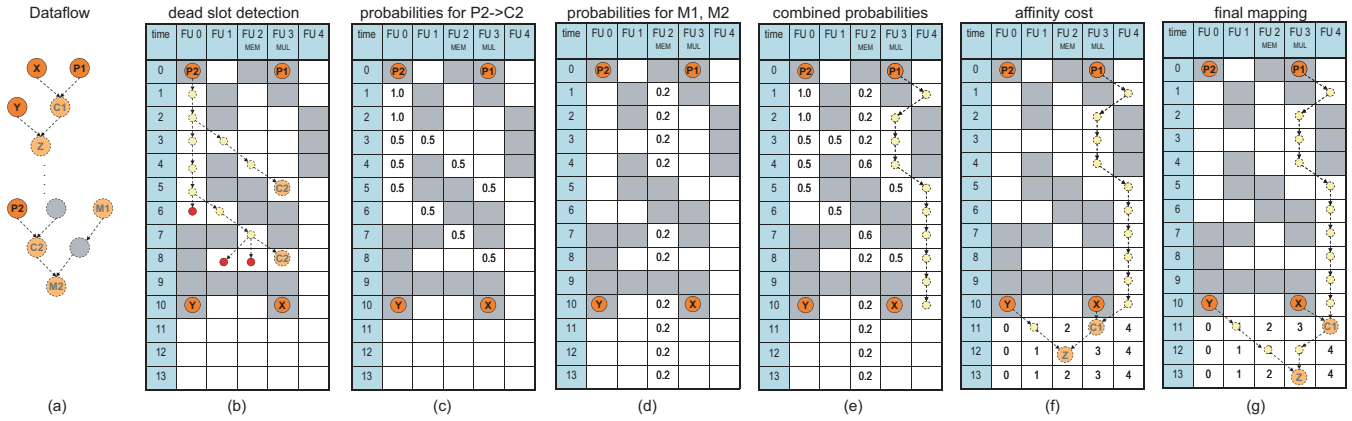

Figure 9: Routing cost calculation example: (a) dataflow graph, (b) - (g) reservation table with computed routing costs.

4.2.3 Finding the Target

Once all routing costs are updated, the router will start finding a path from the source to the target operation. Starting from a slot that contains the source operation, the router visits neighboring slots in the CGRA using a maze routing technique. Each neighboring slot is put into a priority queue and the router visits the slots in order of their routing costs as calculated above.

When a collapsed edge is routed, the router ensures that it finds a path that goes through at least as many FUs as the number of collapsed nodes, so that the collapsed nodes can be expanded later into those FUs. A similar approach is taken for high-fanout edges. Because the high-fanout edges are scheduled with low priority, the corresponding values are likely to have long lifetimes. Therefore, when high-fanout edges are routed, the scheduler attempts to find a path that goes through a register file.

If the target is already placed, the route is towards the slot that contains the target operation. Otherwise, it will find a slot that can execute the target operation. Once a slot is found, the scheduler checks if other edges connected to the target need to be placed, and recurses to route those edges. When an edge has a dependency on other edges as described in Section 3.4, the routing is deferred until all edges in more critical paths are scheduled. When all of the edges are successfully routed, the scheduler moves on to the next operation in priority order.

When the scheduler places recurrence cycles, edges are placed even if their target operations are not placed yet. By calling the router function recursively for all operations in the cycle, the scheduler can put more effort into finding a legal mapping for the recurrence cycles. To prevent exponential compile time for large recurrence cycles, the number of recursive calls is limited to a fixed value. When the scheduler successfully routes all the connected edges, it finalizes the placement of the target operation and proceeds with the next one.

4.2.4 Routing Example

Figure 9 shows an example of how EMS routes an edge with updated routing costs for each slot. Again, we assume no register files in the target architecture for illustration purposes. The DFG in Figure 9(a) is mapped onto the 1x5 CGRA. Here, we assume that P1, P2, X, and Y are already placed and the scheduler is about to route the edge from P1 to C1. Further, C2 is a multiply operation and can only execute on FU 3, and M1 and M2 are memory operations and can only execute on FU 2. First, the scheduler calculates probabilities of routing slots generated for the unplaced edge from P2 to C2 (Figure 9(b)). Then, it identifies dead slots that will not lead to any compatible slots for C2 , as indicated by dark small dots in Figure 9(b). Once all the dead slots are identified, probabilities are propagated along the routing live slots. Figure 9(c) shows the final probabilities. Slot (0,2) gets 1.0 since there is only one path from P2. Slots (0,3) and (1,3) get the probability of 0.5 since there are two routing options from the previous slot.

Next, probabilities are generated for the expensive operations, M1 and M2, that are not placed (Figure 9(d)). With two expensive operations and 10 available slots on FU 2, each slots gets a 0.2 probability.

The probabilities in Figure 9(c) and Figure 9(d) are combined using Equation 5 resulting in Figure 9(e). Based on the probabilities calculated for unplaced edges and nodes, the router finds a path for the edge from P1 to C1 as shown in Figure 9(e). There are two candidate slots for C1; slot (3,11) and slot (4,11). Since C1 and Y have a common consumer Z, the placement of C1 can affect the number of routing resources used later when the edge from Y to Z is routed. As shown in Figure 9(f) and (g), slot (3,11) is preferred to slot (4,11) when considering the common consumer Z. EMS utilizes the affinity heuristic [18] to make this decision. For each slot, the affinity cost is assigned in a way that a higher cost is given as the distance from Y increases. Therefore, the scheduler prefers slots that are close to Y and (3,11) is selected. Later when Z is scheduled, the routing cost can be reduced since Y and C1 are placed close to each other.

4.2.5 Register Constraints

In CGRAs, values with long live ranges can be more efficiently routed through distributed register files. The scheduler must carefully manage register resources so that values stored in the register file are successfully routed to consumers. Traditionally, register allocation is performed after scheduling, and spill code is inserted when the register requirement exceeds the register file capacity. Spilling in the CGRA is quite costly since it involves routing to/from the memory units and may require complete rescheduling of the loop. Moreover, spilling can easily happen due to the small size of the register files.

EMS performs register allocation during scheduling to avoid spilling and guarantee routability through the register files. Register allocation occurs frequently, as it is needed whenever the router visits a register file. So, a simple and fast allocation scheme was

developed that focuses on the routability of stored values. Since EMS gives low priority to high-fanout edges, consumers of the same value are typically scheduled in different times. The scheduler needs to ensure that values stored in register files can be routed to all of their future consumers. The details are omitted in this paper due to space constraints.

4.3 Postpass Steps

When EMS finds a legal schedule, it generates the contents of the CGRA's configuration memories. First, it expands the collapsed operations onto the FU slots that were found. Then, control bits for the routing and computation resources are generated, including MUX selection bits, FU opcode bits, and register file addresses.

5. EXPERIMENTAL RESULTS

5.1 Experimental Setup

To evaluate the performance of EMS, we took 214 loops from four media applications from the embedded domain (H.264 decoder, 3D graphics, AAC decoder, and MP3 decoder). The loops, varying in size from 4 to 142 operations, were mapped onto different CGRA configurations.

The target CGRA architecture is a 4×4 heterogeneous array as shown in Figure 1. Functionality for memory access is limited to 4 FUs and multiplication to 6 FUs. The array contains a 64 entry (16 of which are rotating) central RF with 8 read and 4 write ports wherein only FUs in the first row can directly read/write. All other FUs can only read from the central RF via column buses. The central RF is primarily used for storing live-in values from the host processor. Each FU has its own local RF consisting of 8 rotating register with one read and one write port. Local RFs can be also written by FUs in diagonal directions (upper right/upper left/lower right/lower left). For example, local RF in PE 5 can be written by FUs 0, 2, 5, 8 and 10 and only FU 5 can read from it.

We created three architecture instances by differentiating FU and RF connectivity: mesh-plus, mesh-only and no-RF-sharing. In mesh-plus, FUs are connected in a mesh network, meaning that each FU is connected to its immediate neighboring FUs. Additionally, FUs that are two hops apart are also connected. This is a similar configuration to ADRES [14]. In the mesh-only configuration, FU connectivity is limited to a simple mesh network. The no-RF-sharing configuration has same FU connectivity as mesh-only, but local RFs are not shared by FUs in diagonal directions, meaning that each RF can be written/read only by the neighbouring FU.

The performance and compile time of EMS were compared to three different modulo scheduling techniques: **IMS**: traditional iterative modulo scheduler that does not consider routing efficiency; **NMS**: node-centric modulo scheduler that employs the same heuristics as EMS, but scheduling is conducted in a node-centric way; and, **DRESC**: IMEC's simulated annealing based modulo scheduler. All evaluations were taken on an Intel Core 2 Duo system running at 2.66GHz with 2GB memory. Compile time was measured by using only one core of the system. Scheduling results were verified with a cycle accurate simulator.

5.2 Results

In modulo scheduling, MII defines the theoretical upper bound of the performance of the scheduled loop. Therefore, we calculated the performance of the modulo scheduler by dividing MII by the achieved II in each loop. The performance comparison of the four different modulo scheduling techniques is shown in Figures 10, 11, and 12 for the mesh-plus, mesh-only, and no-shared-RF configurations, respectively. The first four groups show the performance

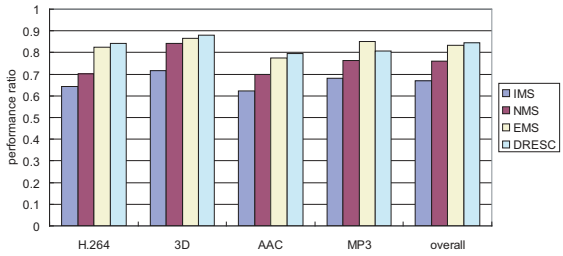

Figure 10: Performance comparison of scheduling strategies for the mesh-plus architecture. The fraction of the theoretical maximum performance is plotted.

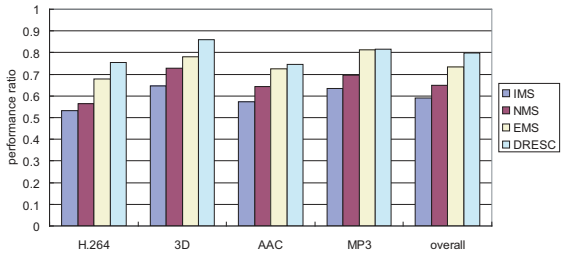

Figure 11: Performance comparison of scheduling strategies for the mesh-only architecture.

results of the loops within each domain and the last group shows the overall performance across all 214 loops.

A more detailed view of the performance comparison between EMS and DRESC is presented in Figure 13 for the mesh-plus configuration. The x-axis shows all 214 target loops grouped by application. Within each application, loops are sorted by increasing MII. The gray line shows the value of MII for each loop. The achieved II for EMS is shown as solid circular dots. The achieved II for DRESC is shown only when it differs from EMS's achieved II, as a vertical line extending from the dot. For the mesh-plus architecture, EMS achieves an average ILP of 9.6 across all the loops.

The final measurement performed is compilation time. The total compile time of all 214 loops for each scheduling technique is shown in Table 1.

5.3 Analysis and Discussion

Comparison with IMS. EMS always outperforms traditional IMS by more than 25% for both mesh-plus and mesh-only configurations. Even though IMS works quite well for conventional VLIWs, the lack of a global resource management strategy causes frequent routing failures which forces II to be increased.

Comparison with NMS. EMS and NMS share most of the heuristics developed in this paper, such as the various cost metrics, stage reassignment, and the reduced dataflow graph. However, EMS achieves 10-13% performance increase while compile time was reduced by 27-46% compared to NMS. This demonstrates the benefits of the edge-centric over the node-centric approach in both performance and compile time measures, as illustrated in Section 3.1.

arch	IMS	NMS	EMS	DRESC
mesh-plus	655	2105	1185	22341
mesh-only	1122	3046	2228	48035

Table 1: Compile time comparison (in seconds).

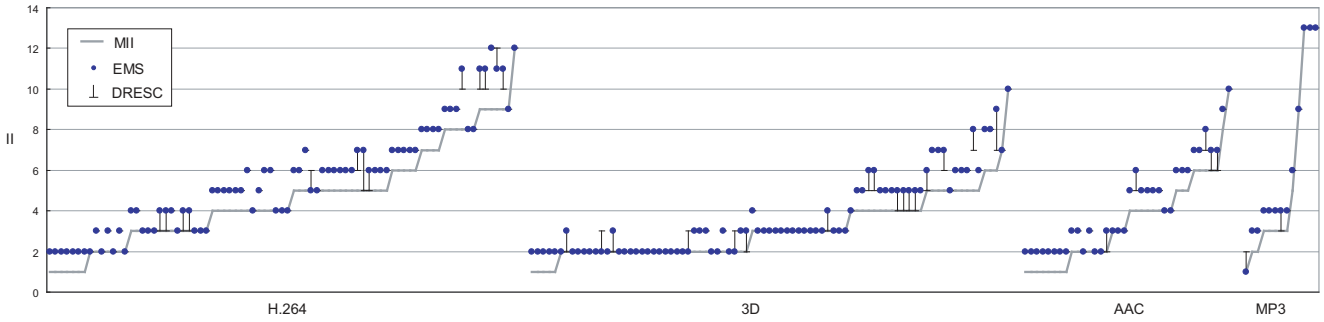

Figure 13: Performance comparison of EMS and DRESC for the mesh-plus architecture.

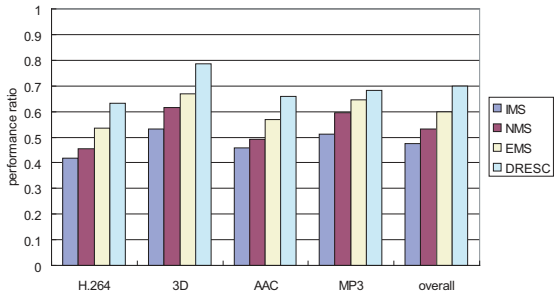

Figure 12: Performance comparison of scheduling strategies for the no-RF-sharing architecture.

Comparison with DRESC. DRESC consistently achieves the best IIs for most of the applications, except MP3 in the mesh-plus architecture. Simulated annealing is an effective strategy for CGRA scheduling, but its high performance comes at the cost of slow compile time. When compared to DRESC, EMS shows quite competitive performance results, achieving 98% and 91% of DRESC's overall performance for mesh-plus and mesh-only architectures, respectively.

For the mesh-plus architecture, EMS shows virtually the same performance as DRESC, achieving the same II or better for more than 85% of loops (Figure 13). For most of the loops that are scheduled at higher IIs, the large number of live-ins was the bottleneck for EMS. Since all of the live-ins are stored in the central RF, there is high contention for central RF ports among the operations that consume live-ins. Though EMS reserves these high contention resources by calculating probabilities in advance, it still fails to achieve the same II as DRESC when the contention is too high.

For the mesh-only architecture, EMS does not perform as well, especially for H.264 and 3D. Those two domains have many communication patterns in which one producer feeds multiple consumers. The execution of such communication patterns is significantly limited with the sparse interconnect in the array. This trend is more obvious when looking at the results of no-RF-sharing configuration 12. EMS is achieving 85% of DRESC's performance when interconnected further reduced by removing shared links to local RFs. This result shows that EMS is more vulnerable to a lack of routing resources. We are currently investigating CGRA designs that have low hardware cost but still enable EMS to achieve high performance.

Compile time. Since there are no intelligent heuristics for global management of routing resources in IMS, it shows the fastest compile time among the four scheduling techniques. Except for IMS, EMS performs the fastest, showing more than 18x speedup over DRESC. A systematic approach for placement and routing indeed allows a reasonable compile time while achieving competitive performance. Compile times for mesh-only are larger than mesh-plus because the achieved IIs are usually higher. Since the scheduler starts at the MII for each loop, it takes more time to get to the solutions with higher IIs.

Effectiveness of Heuristics. EMS employs various heuristics to guide the scheduler towards intelligent routing. The effectiveness of individual heuristics varies based on the application characteristics. The probability heuristic is effective for loops that have high contention on limited resources such as central RF ports or memory slots. Prioritizing edges based on the edge dependency analysis effectively schedules loops with large recurrence cycles, especially when there are many recurrence cycles and some nodes are included in multiple cycles. Stage-reassignment is effective when DFGs have narrow and tall shapes.

6. RELATED WORK

Architectures. Many CGRA-like designs have been proposed in the literature. The designs have different scalability, performance, and compilability characteristics as discussed in Section 2.1. The ADRES architecture [14] is an example of an 8x8 mesh of processing elements with both individual and central register files. MorphoSys [13] is another example of an 8x8 grid with a more sophisticated interconnect network; each node contains an ALU and a small local register file. In the RAW architecture [22], each node is actually a MIPS processor, including memory, registers, and a processor pipeline. In addition, there are both dynamic and static routing networks. PipeRench [7] is a 1-D architecture in which processing elements are arranged in stripes to facilitate pipelining. RaPiD [3] consists of heterogeneous elements (ALUs and registers) in a 1-D layout, connected by a reconfigurable interconnection network. ElementCXI [5] and Ambric [8] are commercialized architecture platforms that present large-scale CGRAs targeting embedded domain applications. Hundreds of computing nodes are connected in hierarchical interconnects and they exploits ILP and TLP available in target applications.

Compilation Techniques. Many techniques have been proposed for compiling to CGRAs. Lee et al. [10] propose a compilation approach for a generic CGRA. They generate pipeline schedules for innermost loop bodies so that iterations can be issued successively. The main focus of their work is to enable memory sharing between operations of different iterations placed on the same processing element. Our work proposes a generic scheduling strategy, and memory sharing and other such optimizations can be integrated into our

system as a preprocessing step. [1] investigated a loop-scheduling problem in CGRA by dividing it into covering, partitioning and layout subproblems. It spatially partitions the CGRA and maps each loop iteration onto the partitioned CGRA. Modulo scheduling differs from this approach in that it time-multiplexes the array for different loop iterations.

RAWCC [11] tackles the scheduling problem for the RAW architecture where all the communication is fully exposed to the compiler. The scheduling problem is broken down into two tasks: spatial assignment and temporal assignment. Operations are placed in each tile first, and time slots are assigned for operations in each time. Convergent scheduling [12] is another compiler technique proposed as a generic framework for instruction scheduling on the RAW architecture. Their framework comprises a series of heuristics that address independent concerns like load balancing, communication minimization, etc. [16] and [2] were also proposed for instruction scheduling of tiled architectures. The scheduling problem in tiled architectures is quite similar to our problem in that the compiler has to manage communications explicitly among computation resources. The main difference is that tiled architectures usually have a dynamically routed network that can sustain some level of routing congestion during runtime. Having no such routing network in CGRAs, the scheduler is responsible for orchestrating every communication so that no congestion occurs. Whereas [11], [12], [16] and [2] focus on ILP and propose scheduling methods for acyclic regions of code, we focus on loop level parallelism. The work of Mei et al. [14] is closest to our work, as discussed in Section 1.

Similar to CGRAs, clustered VLIW machines are also spatial architectures. Much work has been done towards compiling for clustered VLIW machines [6, 17, 21]. Although some of the concepts from these works can be adapted for CGRA compilation, they do not consider the issue of routing values through the sparse interconnection network, which is a crucial step. The measure of affinity used in our scheduler is similar to that used in Krishnamurthy's affinity-based clustering [9].

Stage scheduling [4] re-assigns operations' stages to minimize register pressure for modulo scheduled loops. While stage scheduling is applied as a post pass, EMS re-assigns stages during the modulo scheduling process.

7. CONCLUSION

This paper proposes edge-centric modulo scheduling, an effective modulo scheduling technique for CGRAs. The distributed nature of CGRAs, including sparse interconnect and distributed register files, presents difficult challenges to a compiler. EMS focuses primarily on the routing problem, with placement being a by-product of the routing process. Various routing cost metrics were introduced to give a global perspective of resource management to the scheduler. Edges in the dataflow graph are categorized based on their characteristics and EMS uses different strategies to route them. Overall, EMS improves performance by 25% over traditional modulo scheduling and achieves 85-98% of the performance compared to a state-of-the-art simulated annealing technique. EMS also reduces compilation time by 18x compared to simulated annealing. Experimental results show that the performance of EMS heavily depends on the characteristics of loop structure as well as the underlying CGRA architecture. This encourages an in-depth analysis of the application and exploration of the architecture in the future.

8. ACKNOWLEDGMENTS

Thanks to Greg Steffan and the anonymous referees who provided excellent suggestions for improving the quality of this work. This research was supported by Samsung Advanced Institute of Technology, the National Science Foundation grants CNS-0615261 and CCF-0347411, and equipment donated by Hewlett-Packard and Intel Corporation.

9. 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 *Proc. of the 2006 Design, Automation and Test in Europe*, pages 363–368, Mar. 2006.

[2] K. Coons, X. Chen, S. Kushwaha, K. McKinley, and D. Burger. A spatial path scheduling algorithm for edge architectures. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–140, Oct. 2006.

[3] C. Ebeling et al. Mapping applications to the RaPiD configurable architecture. In *Proc. of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 106–115, Apr. 1997.

[4] A. E. Eichenberger and E. S. Davidson. Stage scheduling: A technique to reduce the register requirements of a modulo schedule. In *Proc. of the 28th Annual International Symposium on Microarchitecture*, pages 338–349, Nov. 1995.

[5] ElementCXI. http://www.elementcxi.com.

[6] J. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, Cambridge, MA, 1985.

[7] S. Goldstein et al. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proc. of the 26th Annual International Symposium on Computer Architecture*, pages 28–39, June 1999.

[8] A. M. Jones and M. Butts. Teroops hardware: A new massively-parallel mimd computing fabric ic. In *IEEE 18th Hot Chips Symposium*, pages 32–41, Aug. 2006.

[9] G. Krishnamurthy, E. Granston, and E. Stotzer. Affinity-based cluster assignment for unrolled loops. In *Proc. of the 2002 International Conference on Supercomputing*, pages 107–116, June 2002.

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

[11] W. Lee et al. Space-time scheduling of instruction-level parallelism on a RAW machine. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, Oct. 1998.

[12] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe. Convergent scheduling. In *Proc. of the 35th Annual International Symposium on Microarchitecture*, pages 111–122, 2002.

[13] G. Lu, H. Singh, M.-H. Lee, N. Bagherzadeh, F. J. Kurdahi, and E. M. C. Filho. The MorphoSys parallel reconfigurable system. In *Proc. of the 5th International Euro-Par Conference*, pages 727–734, 1999.

[14] B. Mei et al. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proc. of the 2003 Design, Automation and Test in Europe*, pages 296–301, Mar. 2003.

[15] B. Mei, F. Veredas, and B. Masschelein. Mapping an H.264/AVC decoder onto the ADRES reconfigurable architecture. In *Proc. of the 2005 International Conference on Field Programmable Logic and Applications*, pages 622–625, Aug. 2005.

[16] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers. Instruction scheduling for a tiled dataflow architecture. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 141–150, Oct. 2006.

[17] E. Nystrom and A. E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proc. of the 31st Annual International Symposium on Microarchitecture*, pages 103–114, Dec. 1998.

[18] H. Park, K. Fan, M. Kudlur, and S. Mahlke. Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures. In *Proc. of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 136–146, Oct. 2006.

[19] M. Quax, J. Huisken, and J. Meerbergen. A scalable implementation of a reconfigurable WCDMA RAKE receiver. In *Proc. of the 2004 Design, Automation and Test in Europe*, pages 230–235, Mar. 2004.

[20] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, Nov. 1994.

[21] J. Sánchez and A. González. Modulo scheduling for a fully-distributed clustered VLIW architecture. In *Proc. of the 33rd Annual International Symposium on Microarchitecture*, pages 124–133, Dec. 2000.

[22] M. B. Taylor et al. The Raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2):25–35, 2002.