Fast Modulo Scheduler Utilizing Patternized Routes for Coarse-Grained Reconfigurable Architectures

WONSUB KIM, Samsung Electronics

YOONSEO CHOI and HAEWOO PARK, Samsung Advanced Institute of Technology

Coarse-Grained Reconfigurable Architectures (CGRAs) present a potential of high compute throughput with energy efficiency. A CGRA consists of an array of Functional Units (FUs), which communicate with each other through an interconnect network containing transmission nodes and register files. To achieve high performance from the software solutions mapped onto CGRAs, modulo scheduling of loops is generally employed. One of the key challenges in modulo scheduling for CGRAs is to explicitly handle routings of operands from a source to a destination operations through various routing resources. Existing modulo schedulers for CGRAs are slow because finding a valid routing is generally a searching problem over a large space, even with the guidance of well-defined cost metrics. Applications in traditional embedded multimedia domains are regarded as relatively tolerant to a slow compile time in exchange for a high-quality solution. However, many rapidly growing domains of applications, such as 3D graphics, require a fast compilation. Entrances of CGRAs to these domains have been blocked mainly due to their long compile time. We attack this problem by utilizing patternized routes, for which resources and time slots for a success can be estimated in advance when a source operation is placed. By conservatively reserving predefined resources at predefined time slots, future routings originating from the source operation are guaranteed. Experiments on a real-world 3D graphics benchmark suite show that our scheduler improves the compile time up to 6,000 times while achieving an average 70% throughputs of the state-of-the-art CGRA modulo scheduler, the Edge-centric Modulo Scheduler (EMS).

Categories and Subject Descriptors: C.1.3 [Processor Architectures]: Other Architecture Styles

General Terms: Architecture, Compiler, Performance

Additional Key Words and Phrases: Coarse-grained reconfigurable architecture, software pipelining, modulo scheduling

ACM Reference Format:

Kim, W., Choi, Y., and Park, H. 2013. Fast modulo scheduler utilizing patternized routes for coarse-grained reconfigurable architectures. *ACM Trans. Architect. Code Optim.* 10, 4, Article 58 (December 2013), 24 pages. DOI: <http://dx.doi.org/10.1145/2555289.2555314>

1. INTRODUCTION

The embedded computing systems in today's portable devices demand high performance and energy efficiency. The complexity of the embedded applications is growing with convergence of different functionalities, such as voice/data communication, high-quality audio, video, and image processing. Coarse-Grained Reconfigurable Architectures (CGRAs) are gaining more interest in the area of domain-specific computing because they offer the potential for high computing throughput with low cost and energy.

Authors' addresses: W. Kim, Samsung Electronics DMC R&D Center, Samsung Electronics 416 Maetan-dong, Yeongtong-gu, Suwon-si, Gyeonggi-do, Korea; Y. Choi (corresponding author) and H. Park, Samsung Advanced Institute of Technology, Samsung Electronics San 14, Nongseo-dong, Giheung-gu, Yongin-si, Gyeonggi-do, Korea.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481 or permissions@acm.org.

© 2013 ACM 1544-3566/2013/12-ART58 \$15.00

DOI: <http://dx.doi.org/10.1145/2555289.2555314>

CGRAs generally consist of an array of a large number of Functional Units (FUs) connected by an interconnect network. Registers are distributed throughout FUs, and the degree of sharing can differ from one implementation to another. The key challenge in deploying CGRAs is compiler scheduling technology that can efficiently map software solutions to hardware so that high performance from many FUs can be achieved. To make good use of CGRA, modulo scheduling of loops is generally employed. There have been research efforts for efficient modulo scheduling technology for CGRA, which are proven to be useful in application domains, such as audio/video and image processing including MP3 and H.264 decoding [Mei et al. 2002; Park et al. 2008; Oh et al. 2009; Kim et al. 2012].

Although the strength of CGRAs is clear in embedded multimedia domains, their long compilation time is a visible weakness, which adversely affects time-to-market cost and developing productivity. Moreover, a slow compilation is one of the main obstacles to wider acceptance of it by other domains. To become a multipurpose accelerator in various domains including web browsing, gaming, user interfaces, and 3D graphics, a fast compilation is a requisite.

The traditional CGRA modulo scheduler, DRESC, defines the CGRA modulo scheduling problem as a tightly coupled Placement and Routing (P&R) problem with modulo constraints and asymmetric routing resources in 3D space [Mei et al. 2002]. It took the simulated annealing approach to tackle the combination of those NP-hard problems. Random placements followed by explicit routing stages are systematically taken using cost functions similar to the well-known PathFinder algorithm in the FPGA community [McMurchie and Ebeling 1995; Ebeling et al. 1995]. A state-of-the-art CGRA modulo scheduler, the Edge-centric Modulo Scheduling (EMS) [Park et al. 2008], gives priority to routing over placement with an observation that routing is a key step in CGRA modulo scheduling. It improves the compilation time by 18 times with a slight performance degradation, compared to the simulated annealing approach. However, its compilation time is still exponentially proportional to the code size since finding a valid routing is generally a search problem even with the guidance of well-defined cost functions. When a new placement option is taken, the routing option between the source and destination resources should be re-examined over a large candidate space. These searching efforts are not additional costs for finding better solutions, but necessary costs for one valid solution.

We propose a fast modulo scheduler for CGRAs, which improves the compile times by multiple orders of magnitude. Our scheduler circumvents the problem of explicit routings coupled with placements in CGRA modulo scheduling by utilizing predefined useful routing patterns. The routing step is intelligently simplified so that it can be integrated into the placement step: a placement option is taken only if its immediate later routings are ensured to be successful. Since a limited set of routing patterns are utilized, the resources and their time slots necessary for guaranteeing future routings can be conservatively categorized into a small number of combinations. By checking on these combinations, whether a routing will be possible is quickly discernable. Our scheduler makes a reservation on resources in advance to ensure future routings. Any unused resources in the actual routing are reclaimed back for remaining routing requirements. This conservative reservation ahead of actual routing is applied only to the routing requirements immediately following the placement of an operation. Since operations are scheduled in a top-down manner, the scheduler has no need to look into routing options far away from the currently scheduled operation.

To verify the usefulness of routing patterns we selected, we observed how many of the routings in existing CGRA schedulers are classified into our qualified patterns. Figure 1 shows the ratio of FU to FU routings that follow our qualified patterns in

Fig. 1. Ratio of qualified paths in FU to FU routings from EMS [Park et al. 2008] on Taiji benchmark [RightWare 2011].

results from EMS [Park et al. 2008]. Over 77 percentages of FU to FU routes are through our qualified paths.

To the best of our knowledge, our technique is the first CGRA modulo scheduling technique that can effectively converge to a polynomial time complexity. We took a real-world benchmark suite in the domains of 3D graphics, which is known to be extremely intolerant to a long compile time. The 3D graphics standards demand run-time compilation capability [Peeper and Mitchell 2003; Lalonde and Schenk 2002]. On this benchmark, our scheduler is faster by 300 times on average, and up to 6,000 times within the average performance degradation of 30% compared to EMS. In some programs, the performance loss is none. The contributions of this article can be summarized as follows.

- We characterize patterns of routings useful to simplifying the routing step in scheduling our target CGRA.
- We propose a fast modulo scheduling technique where future routings are ensured in the placement step.
- We provide an analysis of time complexity showing that the proposed algorithm can effectively converge to a polynomial time complexity for our target CGRA.
- We verified the speed of our compilation technique using a real-world 3D graphics benchmark, Taiji Girl in OpenGL ES 2.0 [RightWare 2011].

The remainder of this article is organized as follows: Section 2 presents the background and motivation of the proposed scheduling technique. Section 3 explains the proposed algorithm in detail. Section 4 provides an example procedure of the proposed algorithm. Section 5 shows the experimental results over them. Section 6 presents related work. Section 7 concludes this article.

2. BACKGROUND AND MOTIVATION

2.1. Target CGRA Architecture

A CGRA consists of an array of FUs, each of which executes word-level operations, communicating through interconnect networks. Generally, a CGRA has three characteristics: number of FUs, sharing of Local Register Files (LRFs), and interconnection topology. For example, a 4×4 array of FUs connected through a mesh interconnect,

where each FU is attached with a small LRF, is an instance of CGRA. Not all FUs have to be homogeneous.

Our target CGRA in this article has a hierarchical layer. It is a 4×4 array of heterogeneous FUs, where four FUs are logically grouped into an FU cluster, organized into four FU clusters. FU clusters are connected to each other by a scalable inter-cluster communication network. For ease of understanding, we call the FU cluster, to which an FU belongs, the *home* FU cluster, whereas we call other FU clusters *remote* FU clusters. An FU can directly send (receive) data to (from) another FU in its home FU cluster. A direct access here means the access path is only composed of transmission nodes, not of any FU or any register file. A transmission node (TRN) is in charge of routing values: Multiplexers and on-pathway flip-flops are examples.

An FU cluster has LRFs that can be directly accessed by FUs in the FU cluster, but each of them does not have dedicated read (and write) ports to LRFs. An FU cannot read an LRF of a remote FU cluster without help of an FU in the remote FU. In other words, data in a remote LRF can only be moved to the FU by an FU in the remote FU cluster. Conversely, an FU can write to a remote LRF through an Inter-Cluster Channel (ICC) in its home FU cluster. An ICC of an FU cluster is a unidirectional channel through which an FU in the FU cluster can broadcast its output to all FUs and LRFs in all other remote FU clusters. An ICC is also a type of TRN. An FU cluster may have more than one ICC. In addition to LRFs, our CGRA has central register files primarily for storing live-ins to loops. Like many other CGRAs, our CGRA is equipped with a small configuration memory containing per-cycle control signals to all components (e.g., TRNs, FUs, and RFs).

Similar to CGRAs, clustered VLIW machines are also spatial architectures. However, from the compiler point of view, our CGRA is differentiated in that various combinations of bypasses and data copies between source and destination FUs need to be explicitly explored to find valid routings. In a clustered VLIW, the data movements are mostly implicitly guaranteed through reads from (and writes to) multiported register files. Most of remaining concerns about the compilers are limited to explicit or implicit data copies among individual clusters. In addition, a clustered VLIW instance composed of more than two clusters is not common.

We use this target CGRA throughout this article. Many details in the specifications, such as FU functionalities, the numbers of total FUs and FU clusters, and the number of ICCs and LRFs per FU cluster, are configurable. The applicability of our algorithm is not limited to the CGRA instance covered in this article. As long as qualified routing patterns can be properly defined for a CGRA, our scheduling technique can be utilized.

2.2. Challenges for Fast CGRA Modulo Scheduling

Modulo scheduling is a software pipelining technique to increase Instruction-Level Parallelism (ILP) by overlapping successive loop iterations [Rau 1994]. The goal is to find a valid schedule that starts a new iteration every Initiation Interval (II). Modulo scheduling for CGRA architecture is known to be an intractable problem. Different from general VLIW architecture, the data from a producer to a consumer FU are explicitly routed through various architectural components, including register files, TRNs, and FUs. These routings are required to be scheduled with modulo constraints.

Consider five example routing paths between two FUs illustrated in Figure 3. These are valid paths in CGRA architecture shown in Figure 2. Numbers below resource names represent time slots in which the corresponding resources are occupied. Suppose the producer operation is mapped on FU2 at time slot 0 and the scheduler is considering placing the consumer operation on FU11 at time slot 100. Assume path3 is chosen for routing because all components in it were available at the specified time slots. If the consumer operation is required to proceed to the next time slot, 101, because FU11 is not

Fig. 2. Target CGRA overview.

Fig. 3. Example routing paths between FU2 and FU11 from CGRA in Figure 2.

available at time slot 100, path3 cannot be automatically reused: for example, consider moving the end time of LRF0 and time of FU3 in path3 ahead by one. FU3, which is available at time slot 98, may be no more available at time slot 99. Consequently, the scheduler should reconsider other routing possibilities, not just path3, but including all other paths shown in Figure 3.

A new placement option inevitably causes a search space exploration for valid routings. The existing CGRA modulo scheduler employed meta-heuristics or cost-based search algorithms to attack this problem. It finds high-quality solutions at the expense of compile times.

To avoid a long compile time, we simplify routing stages in CGRA modulo scheduling so that it can be effectively integrated into the placement step. A placement option is taken only if it is guaranteed not to be invalidated due to later routings. Our scheduler attains this trait by using limited routing patterns. The patterns used by our scheduler have the following characteristics: first, the availability of all resources within a routing path is quickly verifiable. Second, a routing path is elastic so that it can easily support

```

/* Include headers */

void Kernel_P_PixelShader_21(
    Draw_Context *body, Fragment_Batch *batch_in
    Fragment_Batch_Out *batch_out, GLsizei *count) {
    :
    texture_scale0 = body->UniformArray[0][0];
    normal_map_scale0 = body->UniformArray[1][0];
    :
    #pragma CGRA
    for (i = 0, j = 0; i < count; ++i) {
        /* Shader main */
        :
        vv00 = batch_in[i].VaryingArray[0][0];
        vv01 = batch_in[i].VaryingArray[1][0];
        :
        // Processing
        :
        if (no_result) continue;
        batch_out[j].FragColor = result;
        :
        ++j;
    }
}

```

Fig. 4. A C code generated by the translator from a shader code.

a routing with length $t + \alpha$, when it is available for length t . We name these useful routing patterns for a given CGRA *qualified* patterns. Details on how qualified routings are defined between any two FUs are described shortly in next the section.

2.3. 3D Graphics Applications: Shaders

Since there are many GPUs, it is almost impossible to distribute precompiled shader binaries for every GPU. Rather, shader programs are compiled within a given target GPU device during the loading time of an application. In addition, shader codes can even be generated during the runtime depending on various shading effects and user options. These runtime compilation features are included in graphics programming standard such as OpenGL and DirectX [Peeper and Mitchell 2003; Lalonde and Schenk 2002; Rost et al. 2009]. Thus, the compile speed greatly affects the quality of user experiences.

In the 3D graphics pipeline, vertex and pixel shading stages are programmable parts. In general, per-vertex (pixel) shading behaviors are coded in a shader language such as OpenGL Shader Language. To exploit the abundant FUs in CGRA, we coalesce per-vertex (pixel) codes into a loop through a shader-to-C translator and apply a modulo scheduling to the loop. Modulo scheduling is effective in increasing ILP and hiding latencies from long-latency operations like texture-load operations.

Figure 4 presents an example C code translated from a pixel shader code. In order to exploit the software pipelining scheme, the translator forms a loop, whose iterations process a single input item. Note that a pragma directive is inserted just before the loop

Type	Qualified routing pattern example	Architectural delay (shortest path len.)	Total routing delay	Essential resources	
				LRF	ICC
<i>Intra cluster direct</i>	FU1 $\xrightarrow{2}$ FU2	$d(\text{FU1}, \text{FU2}) = 2$	2	-	-
<i>Intra cluster indirect</i>	FU1 $\xrightarrow{1}$ LRF0 FU2 $\xrightarrow{2}$	$d(\text{FU1}, \text{LRF0}) = 1$ $d(\text{LRF0}, \text{FU2}) = 2$	≥ 4	One LRF in dest. cluster	-
<i>Inter cluster direct</i>	FU1 $\xrightarrow{1}$ FU5 $\xrightarrow{1}$	$d(\text{FU1}, \text{ICC0}) = 1$ $d(\text{ICC0}, \text{FU5}) = 1$	2	-	One ICC in src. cluster
<i>Inter cluster indirect</i>	FU1 $\xrightarrow{1}$ LRF1 FU5 $\xrightarrow{2}$	$d(\text{FU1}, \text{ICC0}) = 1$ $d(\text{ICC0}, \text{LRF1}) = 0$ $d(\text{LRF1}, \text{FU5}) = 2$	≥ 4	One LRF in dest. cluster	One ICC in src. cluster

Fig. 5. Categorization of qualified routing patterns and their representative examples: LWP and LRP represent a write port and a read of an LRF, respectively.

for the compiler to recognize the code portion where the software pipelining scheme is applied to.

3. CORE CONCEPTS

3.1. Utilizing Patternized Routes

Qualified Routing Patterns. First, we take advantage of a *direct path* as often as possible. A *direct path* between a source and a destination FU is defined as the shortest path in terms of the delay of TRNs. It consists only of TRNs without any FU or register file between the two FUs. Second, when a direct path is not available, an *indirect path* is utilized. An *indirect path* contains exactly one LRF between the source and destination FUs. The LRF is an LRF in the home cluster of the destination FU. The path from the source FU to the LRF is the shortest path in terms of TRN delay. Similarly, the path from the LRF to the destination FU is also the shortest path in terms of TRN delay. Note that we do not use an FU as a routing resource in the direct and indirect routing paths defined previously. In other words, an FU does not appear between a source and a destination FU. Direct and indirect paths can be subdivided into *intra*- and *inter*cluster paths, depending on whether the home cluster of the source FU (i.e., source cluster) and that of the destination FU (i.e., destination cluster) are the same or not. A path is an *intra*cluster path if the clusters are the same. Otherwise, the path is an *inter*cluster path. An *inter*cluster path contains exactly one ICC among ICCs in the source cluster.

The direct and indirect paths together should be able to cover most of the routing requirements to schedule a DFG for a given architecture. A direct path minimizes the resource usage for a route, while an indirect path can serve any delay by taking advantage of local rotating registers. We call the direct and indirect paths defined earlier *qualified paths*.

Example qualified paths are shown in the second column of Figure 5. The home cluster ID of FU_i is $i/4$ in this example. LRF_i and ICC_i denote an LRF and an ICC of cluster i . The weight of an edge in a path represents the shortest path length in terms of delay of TRNs between the tail and head resources. For simplicity's sake, we omit many individual TRNs by abstracting them into the nonzero weights. These weights are recapped in column 3. Based on the weights, the total routing delay of a direct (indirect) path can be calculated as shown in column 4. Notice that the routing delay of

Type \ Resource	Resource	Time	
Intra cluster	one write port to LRF(c)	$T(n) + \text{Latency}(n) + d(FU(n), LRF(n))$	$d(FU(n), LRF(n))=1$
Inter cluster	one write port to LRF(c)	$T(n) + \text{Latency}(n) + d(FU(n), ICC(n)) + d(ICC(n), LRF(c))$	$d(FU(n), ICC(n))=1$ $d(ICC(n), LRF(c))=0$
	one ICC(n)	$T(n) + \text{Latency}(n) + d(FU(n), ICC(n))$	

(a)
(b)

Fig. 6. Conservative routing resource reservation for an edge (n, c) : (a) Resources and their time slots to be reserved for a route; (b) $d(s, d)$ denotes the architectural delay from s to d . For an operation n , $LRF(n)$ and $ICC(n)$ are LRF and ICC in the home FU cluster of $FU(n)$. The schedule time and operation latency of n are denoted as $T(n)$ and $\text{Latency}(n)$, respectively.

a direct path is unchangeable. On the other hand, an indirect path has only a minimum delay assuming at least one cycle time is consumed in the LRF. The essential resources for each type of qualified path can be summarized into a fixed set of LRFs and ICCs as shown in columns 5 and 6.

Conservative Reservation. Our scheduler only uses routes that conform to the qualified paths. One of the benefits qualified patterns offer is that most of the routing requirements for a DFG can be supported by these patterns. An indirect path is elastic in that it can serve any routing delay $t + \alpha$ when it serves t , as long as the delay is not smaller than its minimum delay. Another advantage of qualified paths is that the essential resources and their time slots relative to the source and destination FUs can be determined. We utilize these characteristics of qualified paths to ensure successful routings originated from an operation in advance when the operation is placed at one FU. More specifically, the resources and their time slots that can guarantee a route are conservatively estimated and reserved before the actual routing step happens.

The combinations of resources and their time slots to be conservatively reserved can be predefined as in Figure 6(a). The estimation assumes indirect paths are used. The schedule time and operation latency of n are denoted as $T(n)$ and $\text{Latency}(n)$, respectively. A careful comparison of two tables in Figures 5 and 6 reveals how the conservative resource reservations are derived from the categorization of qualified paths. Notice write ports to LRFs are considered in the reservation.

The qualified paths can be extracted in the architecture analysis step, using the Floyd-Warshall shortest-path algorithm. A pair of source and destination FUs can have more than one direct (indirect) path since there could be more than one shortest path with the same delay between them.

3.2. Algorithm Overview

Basically, the proposed scheduler schedules operations in a top-down manner (i.e., a consumer operation is scheduled after the producer operations).¹ Whenever operation n is to be scheduled onto a candidate FU and a time slot, its incoming and outgoing edges are inspected: (1) for an incoming edge, whether the actual routing can be established with a previously reserved routing resource, and (2) for an outgoing edge, whether

¹Constant and live-in values are exceptions and will be described later in Section 3.5.

there is any available FU for the corresponding consumer and whether the resources for guaranteeing routings from the source FU to that FU available are examined. Only if the conditions are successfully satisfied can operation n be scheduled to the given FU and time slot. For ease of understanding, we explain the handling of outgoing edges before incoming edges.

Conservative Reservation for an Outgoing Edge. Given candidate FU, $FU(n)$, and schedule time, $T(n)$, for outgoing edge, (n, c) , to consumer c , corresponding routing resources in Figure 6 should be examined. In other words, at least one instance in every set of the specified types of resources should be available. Notice the FU the consumer will be mapped to, $FU(c)$, differentiates the type of resources. Our scheduler strives to find the combination of $FU(c)$ that will allow conservative reservation of routing resources for (n, c) at corresponding time slots. If more than one such FU is available for c , the one with the minimum cost by the following metric is chosen:

$$cost(FU) = \text{number of operations scheduled in } FU. \quad (1)$$

As a result, the FU of consumer c , $FU(c)$, and routing resources for (n, c) are reserved in advance before consumer c is scheduled, when operation n is scheduled.

Establishing Incoming-Edge Routing. For a given incoming edge, (p, n) , to operation n , the routing of an operand from p to n is almost guaranteed because the required routing resources are conservatively reserved when p is scheduled. The remaining jobs of the scheduler are (1) to determine if a direct or an indirect path is used and (2) to check on if a read port of the reserved LRF is available if an indirect path is chosen. The first decision is depending on the schedule time difference of p and n , which is defined as follows:

$$\text{DiffSchedTime}(p, n) = T(n) + II * \text{Dist}(p, n) - (T(p) + \text{Latency}(p)), \quad (2)$$

where $\text{Dist}(p, n)$ means the dependence distance between p and n . If $\text{DiffSchedTime}(p, n)$ is the same as $\text{DirectPathLen}(FU(p), FU(n))$, a *direct* path between $FU(p)$ and $FU(n)$ is used. $\text{DirectPathLen}(FU(p), FU(n))$ is the length of a direct path from $FU(p)$ to $FU(n)$, which is already determined when routing patterns are extracted. Notice $FU(p)$, $FU(n)$, and $T(p)$ are already fixed. $T(n)$ is the candidate time slot currently being examined. If $\text{DiffSchedTime}(p, n)$ is longer than $\text{DirectPathLen}(FU(p), FU(n))$, an *indirect* path between $FU(p)$ and $FU(n)$ is used. The availability of an indirect path involves the second task described earlier, a checking on an LRF read port. The LRF read ports are not reservable before scheduling n because $\text{DiffSchedTime}(p, n)$ is not calculable without $T(n)$. If a direct path is used, any unused resources such as LRF read ports are retrieved for progressing to the next operation.

The Range of Candidate Time Slots. For operation n , the scheduler scans the range of time slots, checking on their incoming and outgoing edges as described earlier. Candidate time slots for n are calculated using Equation (3). Our scheduler scans the timing range $[\text{MinTime}(n), \text{MaxTime}(n)]$ for $T(n)$, where $\text{MaxTime}(n)$ is $\text{MinTime}(n) + II$. By the time the schedule of n is considered, $FU(p)$ is known since producer p is scheduled already and $FU(n)$ is also reserved in advance when the producers are scheduled.

$\text{MinTime}(n)$

$$= \max_{p \in \text{pred}(n)} \begin{cases} \max(0, T(p) + \text{Latency}(p) + \text{DirectPathLen}(FU(p), \\ FU(n)) - II * \text{Dist}(p, n)), & \text{if } p \text{ scheduled.} \\ \text{MinTime}(p), & \text{otherwise} \end{cases} \quad (3)$$

3.3. Analysis on Algorithm Complexity

Algorithm 1 illustrates the overview of our scheduling process for a given II. We calculate the complexity of the algorithm with the following parameters: the total number of operations in the given DFG (N), the average number of producers and consumers for a node (P and S), the average number of qualified paths for a given pair of a producer and a consumer (Q), the average number of resources to be examined in a given qualified path (R), the average number of FUs that can support a given operation (F), and the number of resources that should be conservatively reserved (C). The complexity of the algorithm sketch for Algorithm 1 can be calculated into $N \times \text{II} \times ((P \times (Q \times R)) + (S \times (F+C)))$. We observed that P and S are bound by small constant values in most cases. For our target CGRA, Q is also bound by small constant values. C is also a constant value, that is, two (LWP and ICC). For the proposed qualified paths, R can also be bound by a small constant. Thus, the calculation boils down to $O(N \times \text{II} \times F)$. The process may repeat up to II iterations. The complexity in total becomes $O(N^3 \times F)$, since $\text{II} \leq N$. F is generally much smaller than the number of total FUs in CGRA because not all FUs are homogeneous and not all FUs are unoccupied for a given time slot. Hence, the algorithmic complexity of Algorithm 1 is $O(N^3)$. In addition, we can minimize the number of attempts of the whole process by increasing the initial II to a value that can achieve a valid schedule in the very near future. The details and experimental results will be given in Section 5. Under this condition, the complexity can be reduced to $O(N^2)$.

ALGORITHM 1: Algorithm overview for a given II

```

for each node  $n$  in DFG do
  for each time slot  $t$  in  $[\text{MinTime}(n), \text{MaxTime}(n)]$  do
    for each producer  $p$  of node  $n$  do
      ScansCandidateQualifiedPaths( $p, n, t$ );
    end
    for each consumer  $c$  of node  $n$  do
      DetermineFU( $c, t$ );
      ReserveConservatively( $n, c, t$ );
    end
  end
end

```

Notice that the architectural analysis phase employing the Floyd-Warshall algorithm with $O(M^3)$ complexity, where M denotes the total number of architectural entities, is applied only once when the compiler is built for the given architecture and not included in our main scheduling process.

3.4. Sharing of Routing Resources

When a single value is required by two or more consumer operations, it is crucial to make the routing paths share the routing resources as far as possible. It helps enhance the schedule quality by saving valuable scheduling resources so that the contentions on them can be minimized. Let us consider two different routes, $FU1 \rightarrow FU5$ and $FU1 \rightarrow FU9$, shown in Figure 7. In this case, the two routes can share $ICC0$ by using paths P1 and Q1. If either pair of paths, (P1, Q2) or (P2, Q1), was chosen, both $ICC0$ and $ICC1$ are occupied. Existing CGRA modulo schedulers often exhibit sharing in routing resources either as a result of simulated annealing or by cost-based search [Mei et al. 2002; Park et al. 2008]. However, it is inevitable for them to employ a search across a large search space in order to include the options of resource sharing. On

Fig. 7. Direct paths from FU1 to FU5 and FU9: ICC0 and ICC1 are two ICCs in the home cluster of FU1.

the other hand, in our scheduler, the sharing of resources is obtained as a by-product of the architectural analysis, which is taken only once when the scheduler is built. A unique ID is given for each architectural component. In the path reconstruction phase of the Floyd-Warshall algorithm, the order of unique IDs is honored so that the resulting shortest paths between a source and a destination are lexicographically ordered.

For example, in Figure 7, a number outside a box denotes the unique ID of the corresponding resource. When FU1 and FU5 are routed, paths $1 \rightarrow 4 \rightarrow 2$ and $1 \rightarrow 5 \rightarrow 2$ are considered in a row. Similarly, for FU1 and FU9, paths $1 \rightarrow 4 \rightarrow 3$ and $1 \rightarrow 5 \rightarrow 3$ are scanned in a row. Thus, if 4 is available, paths $1 \rightarrow 4 \rightarrow 2$ and $1 \rightarrow 4 \rightarrow 3$ are selected.

In a similar fashion, an LRF is shared if the multiple consumers have the same producer in common. This means the scheduler can avoid multiple redundant writing of the same value, saving register entries and write ports to an LRF.

3.5. Additional Concerns

Constants and Live-ins. In our target CGRA, constants and live-ins to a loop are stored in constant buffers and central register files, respectively. Thus, the routing from a constant buffer to an FU and the routings between a central register file and an FU should be included in scheduling decisions. Reading constants or live-ins can be represented as pseudo-operations without any producer to them. Scheduling such operations in a top-down manner incurs unnecessarily long routings. We scheduled them backward by scanning from their ALAP (i.e., as late as possible) time when their consumers are scheduled. An FU is allowed as a routing resource between an FU and a constant buffer (a central register file) in the qualified indirect routing patterns. This is because a copy is required to move values in constant buffers (central register files) to local registers.

Recurrence Handling. Since the shader program is written in a per-element manner, there should be no dependence between the iterations except the basic induction variable for the coalesced loop. Hence, the scheduler has to carefully handle those recurrences, considering the timing constraints.

To cope with the constraints, the proposed scheduler exceptionally adopts a backtracking scheme and makes simple adjustments to the schedule order of the operations within the recurrence. This heuristic is based on the observation that the edge from an FU to a central register has to be placed as a direct path in the given architecture. The scheduler gives priority to an operation writing a value into the register, so that such an operation is handled as early as possible.

4. CASE STUDY

For better understanding of the scheduling algorithm, this section provides a case study with one pixel shader example from the Taiji benchmark [RightWare 2011], whose DFG

Fig. 8. The DFG of a pixel shader in Taiji [RightWare 2011].

is drawn in Figure 8. The explanation covers the scheduling of highlighted operations (i.e. n_{145} , n_{146} , n_{153} , and n_{154}), assuming they are scheduled in this order.

The target CGRA architecture is assumed to be containing 16 FUs as shown in Figure 2. The FUs are grouped into four FU clusters, denoted as FC0, FC1, FC2, and FC3, each of which has two ICCs and one LRF. The LRF is connected to all FUs in the cluster but has six read ports and three write ports, which are shared. Thanks to the heterogeneity of FUs, operation *fadd* can be handled only on *FU1*, *FU5*, *FU9*, and *FU13*, while *fmul* can be handled only on *FU3*, *FU7*, *FU11*, and *FU15*. The latencies of both operations are 4.

Figure 9 illustrates the Modulo Reservation Table (MRT) for FUs and other routing resources, where II is 12. For FUs, both input (“I”) and output (“O”) entries are given, assuming execution stages of all operations are pipelined. The already-occupied entries are marked with “X” or with an operation ID, while the reserved entries are denoted as “r” with an operation ID. For example, “r145” means the reserved entries for operation n_{145} ’s outgoing edges.

Fig. 9. Modulo reservation table (MRT) at $\Pi = 12$. FU cluster 3 is omitted for ease of understanding. LRP[2 5] of FU cluster 0/2 and and time slots [1 52], [60 69] are omitted for ease of display.

The notations used for the descriptions of the scheduling process are summarized as follows. The schedule time of operation n , $T(n)$, once appeared in Section 3.2, denotes the input time of n (i.e., the time for FU input MRT), whereas the output time of n , $Tout(n)$ (i.e., the time for FU output MRT), is calculated as $T(n) + Latency(n)$. The architectural delays used in the routing resource reservations are assumed to be the $d()$ values in Figure 6(b). By folding these notations and architectural delays into the time column in Figure 6(a), the simplified conservative reservation combinations are derived as follows:

- An intraccluster routing for edge (n, c) : one write port to $LRF(c)$ at time $T_{out}(n) + 1$
- An intercluster routing for edge (n, c) : one write port to $LRF(c)$ at time $T_{out}(n) + 1$ and one $ICC(n)$ at time $T_{out}(n) + 1$,

where $LRF(n)$ and $ICC(n)$ represent one of the LRFs and one of the ICCs in the FU cluster of $FU(n)$, respectively.

Operation Mapping and Routing Resource Reservation. Let us consider placing n_{145} first, where n_{145} is booked onto $FU1$. As shown in Figure 9, the scheduler attempts to place it at $T(n_{145}) = 70$, which means $Tout(n_{145}) = 74$ given that the latency of fadd is 4. Figure 9 shows the placement. As mentioned in Section 3.2, the proposed algorithm reserves routing resources, such as ICC and LWP (i.e., write ports to an LRF), in between n_{145} and its consumers. For the reservation, the scheduler finds out the candidate FUs for the consumers. For example, $FU1$, $FU5$, $FU9$, and $FU13$ are the candidates for n_{153} , a consumer of n_{145} . Since n_{145} will be placed onto $FU1$, placing n_{153} onto $FU1$ requires resources for intracluster routing resources, while placing n_{153} onto $FU5$, $FU9$, or $FU13$ requires resources for intercluster routing resources. In this

Fig. 10. Conservative resource reservation for outgoing edges of n_{153} ; shaded boxes are occupied.

case, one of the ICCs in FC0 and each one of the LWPs in all clusters are available at time $75 (= 74 + 1(\text{routing time}))$, and all of the four FUs are valid candidates.

Now the scheduler attempts to determine an FU for consumer n_{153} . In this case, $FU5$ is chosen since it has the minimum cost among four candidates with an arbitrary tie-breaking ($\text{cost}(FU1) = 9$ and $\text{cost}(FU5) = \text{cost}(FU9) = \text{cost}(FU13) = 8$). After determining $FU5$ for n_{153} , the scheduler reserves an ICC in FC0 and an LWP in FC1 at time 75, as marked as “r145” shown in Figure 9. After this reservation is confirmed, the scheduler can place n_{145} at $T(n_{145})$ safely.

In the same manner, the scheduler schedules another producer n_{146} on $FU7$ with $T(n_{146}) = 53$ (i.e., $Tout(n_{146}) = 57$). Since n_{146} is scheduled on the same FU cluster (FC1), where n_{153} is placed, the scheduler reserves only an LWP in FC1 (i.e., not an ICC) at time $58 (= 57 + 1)$ in Figure 9. This reservation is indicated as “r146” in Figure 9.

The Whole Picture for n_{153} . Let us consider placing n_{153} . Note that it is determined to place the operation on $FU5$ in FC1; hence, it is only needed to determine its schedule time, that is, $T(n_{153})$. The MinTime and MaxTime of $T(n_{153})$ are 76 and 87, respectively, as a result of substituting $Tout(n_{145})$ and $Tout(n_{146})$ into Equation (3). Examining both the input and output MRTs of $FU5$ reveals that time slots from 76 to 85 are unavailable for $T(n_{153})$.

Now the scheduler can find out time slot 86, where $FU5$ is available for n_{153} . However, the reservation for a consumer n_{154} is impossible, as shown in Figure 10(a). In the figure, each box represents a routing resource and a time slot pair. The shaded boxes are already occupied ones. A circled number designates a box. ② and ③ are two ICCs and ④, ⑤, and ⑥ are three LWPs. Note that all ICCs are reserved at time 91 ($= 90 + 1$), so that the intercluster paths (Paths 1 and 2) are unavailable. Similarly, all LWPs in FC1 are reserved, and the intracluster indirect paths (Paths 3, 4, and 5) are also unavailable. The scheduler cannot ensure the future routing for n_{154} , a consumer of n_{153} . Hence, the scheduler regards the placement as a failure and attempts to place n_{153} on a different time slot. Let us consider time slot 87 (i.e., $Tout(n_{153}) = 91$), which allows the reservation of an ICC (⑧) in FC1 and an LWP in FC2 (⑩) at time 92 ($= 91 + 1$) in Figure 10(b). Hence, the scheduler reserves the ICC and the LWP at time 92 ($= 91 + 1$) and goes on.

Fig. 11. Establishing routes to n_{153} and confirming its placement.

After making the reservation, the scheduler attempts to route the incoming edges, which are n_{145} to n_{153} and n_{146} to n_{153} . For the former, the timing difference between two operations, that is, $\text{DiffSchedTime}(n_{145}, n_{153})$, is 13, which is larger than $\text{DirectPathLen}(n_{145}, n_{153})$. Hence, the scheduler tries routing the edge through an indirect path ① $\rightarrow$ ③ $\rightarrow$ ⑥ $\rightarrow$ ⑦ $\rightarrow$ ⑧ $\rightarrow$ ⑭ as shown in Figure 11. Though some routing resources, such as ②, ④, and ⑤, are occupied already, an intercluster indirect path is ensured to be available, due to the conservative reservation. ③ and ⑥ are examples of the reserved resources. In the same fashion, edge (n_{146}, n_{153}) can be routed through path ⑨ $\rightarrow$ ⑪ $\rightarrow$ ⑫ $\rightarrow$ ⑬ $\rightarrow$ ⑭ in Figure 11. Since all the incoming edge routings are done, the schedule of n_{153} is completed.

Retrieving Unused Routing Resources. In a similar manner, n_{154} can be placed on FU_{11} at time 93 but through a direct path, ⑦ $\rightarrow$ ⑧ $\rightarrow$ ⑨ in Figure 10. Recall that one ICC(⑧) in FC_1 and one LWP(⑩) in FC_2 at time 92 in Figure 10 are already reserved for a consumer n_{154} when n_{153} is scheduled. However, the LWP(⑩) is not actually used because an intercluster direct path is built. In this case, the scheduler retrieves the LWP(⑩) for further scheduling.

5. EXPERIMENTAL RESULTS

5.1. Experimental Setup

To evaluate the effectiveness of the proposed algorithm, we took 34 shaders from the Taiji benchmark in Basemark ES 2.0 [RightWare 2011]. Each shader code is translated into a C program containing a for-loop as described in Section 2.3. The number of operations for a shader varies from 50 to 380. We implemented the proposed scheduling algorithm within our in-house compiler. The machine used for experiments is an Intel Xeon system with 2.67GHz with 16GB memory where CentOS 5.6 (Linux Kernel 2.6.18) is installed.

The target CGRA is a 4×4 heterogeneous array of FUs and organized into four clusters as described in Section 2.1 and in Figure 2. The FUs are heterogeneous so that load/store operations can be handled by four FUs; floating-point add and multiplication (fadd/fmul) operations are supported by four FUs, respectively. The most complex operation, texture-load, is provided by only one FU. There are two central register files, each of which has 64 entries with four read and four write ports, whereas each of the per-cluster LRFs has 48 entries with six read and three write ports. Note that the local

register files are used as rotating registers. Each cluster has two ICCs for intercluster communication.

We compared the proposed algorithm, called Fast Modulo Scheduler (**FMS**), to the existing state-of-the-art CGRA Modulo Scheduler, the Edge-centric Modulo Scheduler (**EMS**) [Park et al. 2008]. **EMS** improves the compile time by 18 times within 98% throughput of DRESC's simulated annealing approach [Mei et al. 2002]. In addition to **FMS**, we evaluated **VFMS**. **VFMS** is designed primarily for very short compile time, even shorter than that of **FMS**. We chose the starting II of **VFMS** as $(\min II \times \lfloor \frac{\text{number of FUs in an FU-cluster}}{\text{number of ICCs in an FU-cluster}} \rfloor)$, instead of the theoretical minimum II. Note that this initial II value guarantees no contention in ICCs, which is one of the most contended resources in the target CGRA. In the given architecture, that value is $2 \times \min II$. To examine whether it is trivial to find a valid schedule with a doubled minII using the previous approach, we also compared another version of **EMS**, **EMS2MII**, which starts from doubled min II. Our comparisons are threefold: compilation time, performance, and register usage.

5.2. Compilation Time

Since we are targeting shader codes, the compile time is the foremost priority we are pursuing. More specifically, we are aiming at compiling all shader codes for a 3D application within a few seconds.

Table I shows the measured compile times of the Taiji benchmark using **EMS** and **FMS** in columns 10 and 12, respectively. Depending on the size of codes, the compile times of **EMS** range from around 1 to 3,500 seconds. On the other hand, compile times of **FMS** are less than 1 second, varying between 0.01 to 0.9 second. **FMS** is up to 6,074 times faster than **EMS**, 310 times on average. **VFMS** finds solutions extremely fast, as shown in column 13. As shown by the difference between $2 \times \min II$ and the achieved II from **VFMS**, **VFMS** often finds its solution at its first attempt. This reveals that **VFMS** is an effective heuristic when a compile time budget is even tighter for an iterative scheduling approach. **EMS2MII** is faster than **EMS** but it still spends quite a long time, from 1 second to hundreds of seconds. **EMS2MII** finds a solution at its first attempt in most cases but does not reduce the compile time of that attempt. Therefore, its speed-up of compile time over **EMS** is originated from the decreased total counts of trial but not from the reduced complexity of one iteration. This shows that finding a valid modulo schedule for CGRA merely by increasing II is not a trivial problem.

The compile times of 34 shader codes as their code size rise are plotted in Figure 12. Notice that the y-axis is displayed in a logarithmic scale. It is easy to perceive that the compile time of **EMS** is exponentially proportional to the number of operations. As the number of operations increases, the schedule length is generally linearly prolonged, which causes long routings in terms of schedule time. For **EMS**, these long routings are harder to map to CGRA because the search space for them expands exponentially to their lengths. On the other hand, compile times of **FMS** are proportional to the number of operations with polynomial complexity, as analyzed in Section 3.3, which is $O(N^3)$. The trends in Figure 12 verify our algorithms' polynomial time complexity. The bends in the left corner of the curves for **FMS** and **VFMS**, which naturally appear when polynomial complexity curves are projected into logarithmic scale, are more noticeable due to the approximation by curve-fitting software.² To better demonstrate the compile time trends of **FMS** and **VFMS**, we show data points plotted also in linear scale within the range of 0.01 second to 1 second in Figure 13.

²Our experimental precision of compilation time is up to the number in the second decimal. Therefore, the compilation time under 0.01 second (10ms) is rounded up to 0.01 second.

Table I. Comparison of Different Scheduling Strategies
Indices of shaders are sorted in ascending order of the number of nodes (smallest to largest).

Shader name	Id.	Num. Node	Num. Edge	MII (Res./Rec.)	Achieved II				Compilation Time (sec.)				Register Usage		
					EMS	EMS2MI	FMS	VFMS	EMS	EMS2MI	FMS	VFMS	EMS	FMS	VFMS
shadow-depth-skinned	P1	75	93	6 (3/6)	6	12	6	12	1.53	1.93	0.01	0.01	12	15	8
skybox	P2	86	103	6 (3/6)	6	12	6	12	2.68	3.42	0.01	0.01	15	31	16
reflect-alpha	P3	108	130	6 (4/6)	6	12	8	12	3.74	6.93	0.02	0.01	46	43	24
compose-bloom	P4	114	142	6 (6/6)	7	12	10	12	13.60	4.86	0.03	0.01	50	38	32
brdf-alpha	P5	114	136	6 (4/6)	7	12	8	12	9.84	5.59	0.02	0.01	64	34	27
reflect	P6	115	137	6 (4/6)	6	12	8	12	12.60	7.47	0.03	0.01	36	44	27
db-shader-blur2	P7	130	169	9 (9/6)	10	18	12	18	12.60	5.99	0.03	0.01	44	39	29
db-shader-blur1	P8	130	169	9 (9/6)	10	18	12	18	13.40	6.01	0.02	0.02	45	39	29
underwater-alpha	P9	142	181	6 (6/6)	8	12	11	12	31.90	12.10	0.05	0.02	58	66	59
underwater	P10	143	182	6 (6/6)	8	12	11	12	42.10	13.10	0.05	0.02	61	68	60
db-shader-bloom	P11	183	243	13 (13/6)	16	26	18	26	50.30	15.68	0.06	0.01	63	50	33
hair	P12	232	329	16 (16/6)	16	32	24	32	122.34	174.69	0.18	0.03	172	149	114
water	P13	246	337	17 (17/6)	22	34	31	35	1640	279.68	0.27	0.04	181	161	146
brdf	P14	254	355	17 (17/6)	17	34	21	34	82.19	183.01	0.11	0.02	157	153	107
brdf-normalmapped	P15	320	465	23 (23/6)	23	46	33	46	133.58	212.74	0.33	0.04	199	167	112
skin-skinned	P16	379	568	28 (28/6)	29	56	41	56	424	400.66	0.51	0.04	199	174	126
silk-skinned	P17	412	590	24 (24/6)	31	48	46	48	3530	530.41	0.89	0.05	199	172	156
db-shader-blur2	V1	52	73	5 (3/5)	5	10	7	10	0.80	1.16	0.02	0.01	11	18	16
db-shader-blur1	V2	52	73	5 (3/5)	5	10	7	10	0.83	1.20	0.01	0.01	11	18	16
db-shader-bloom	V3	52	73	5 (3/5)	5	10	7	10	0.82	1.18	0.02	0.01	11	18	16
compose-bloom	V4	55	77	5 (4/5)	5	10	7	10	0.91	1.29	0.01	0.01	11	20	16
skybox	V5	98	132	5 (4/5)	5	10	10	10	2.30	2.58	0.03	0.01	42	41	41
reflect-alpha	V6	122	170	5 (5/5)	7	10	13	13	8.08	3.16	0.06	0.03	51	32	32
brdf-alpha	V7	136	188	5 (5/5)	8	10	12	12	12.80	3.38	0.05	0.02	40	40	40
underwater-alpha	V8	140	192	6 (6/5)	7	12	14	14	8.55	4.56	0.07	0.04	44	35	35
reflect	V9	142	194	6 (6/5)	7	12	14	14	7.98	4.77	0.05	0.04	56	34	34
underwater	V10	148	202	6 (6/5)	8	12	13	13	20.70	6.24	0.06	0.02	62	38	38
hair	V11	212	308	11 (11/5)	12	22	21	23	17.60	9.66	0.13	0.03	72	52	47
shadow-depth-skinned	V12	236	383	14 (14/5)	17	28	21	28	91.80	15.83	0.14	0.03	106	90	69
brdf-normalmapped	V13	245	384	12 (12/5)	15	24	24	24	37.80	12.97	0.23	0.04	65	50	50
brdf	V14	293	470	20 (20/5)	20	40	32	40	37.60	28.41	0.28	0.03	72	49	40
water	V15	296	469	19 (19/5)	20	38	32	38	37.20	39.43	0.33	0.03	86	67	58
skin-skinned	V16	368	604	20 (20/5)	27	40	37	40	259	28.12	0.52	0.04	83	88	76
silk-skinned	V17	368	604	20 (20/5)	25	40	37	40	287	28.04	0.52	0.04	74	88	76

Fig. 12. Comparison of compile times: y-axis is in logarithmic scale.

Overall, **FMS** and **VFMS** satisfy the compile time requirement for 3D applications, which is not even approachable with existing CGRA modulo schedulers.

5.3. Performance

The performance ratios of **FMS** and **VFMS** against **EMS** are shown in Figures 14 and 15. Since the performance of a modulo scheduled code is inversely proportional to its initiation interval, Π , the performance degradation is calculated by the following equation:

$$\begin{aligned}
 \text{Performance degradation of } \mathbf{FMS} &= \frac{\text{Performance}_{\mathbf{EMS}} - \text{Performance}_{\mathbf{FMS}}}{\text{Performance}_{\mathbf{EMS}}} \\
 &= \frac{\frac{1}{\Pi_{\mathbf{EMS}}} - \frac{1}{\Pi_{\mathbf{FMS}}}}{\frac{1}{\Pi_{\mathbf{EMS}}}} = \frac{\Pi_{\mathbf{FMS}} - \Pi_{\mathbf{EMS}}}{\Pi_{\mathbf{FMS}}} \quad (4)
 \end{aligned}$$

FMS suffers 30% performance degradation on average compared to **EMS**, which is subdivided into an average 22% for pixel and an average 37% for vertex shader codes. The achieved performance of our faster version, **VFMS**, is lower by 45% compared to **EMS**. The performance loss is expected since our scheduler explores a much smaller search space by confining routing patterns.

The detailed Π numbers including the minimum Π ($\text{min}\Pi$), a theoretical upper bound of the performance of the scheduled loop, are given in columns 5 through 9 in Table I. The two numbers in parentheses in column 5 represent minimum resource Π ($\text{Res}\Pi$) and minimum recurrence Π ($\text{Rec}\Pi$), respectively. As inferred from steady $\text{Res}\Pi$ throughout all shader codes, recurrences in our shader codes share similar patterns in common. Since handling of recurrences is not the primary focus of **FMS**, the result of

Fig. 13. Comparison of compile times between FMS and VFMS: y-axis is in linear scale.

FMS is not so effective in the cases **RecII** is dominant (i.e., $\text{RecII} > \text{ResII}$) as in the opposite cases.

5.4. Differences in Pixel and Vertex Shader Programs

We observed that **FMS** performs differently for pixel and vertex shader codes. While **FMS** is 629 times faster on average than **EMS** within an average 22% performance degradation for pixel shader codes, it is around 153 times faster than **EMS** with a larger performance degradation of 37% for vertex shader codes.

Pixel shader codes have texture load operations whose latency is generally much longer than other operations. These long-latency operations make the total schedule length of the loop long, resulting in long routings. As described earlier, **EMS** spends a lot of time to map these long routings into CGRA, whereas **FMS** is capable of quick mapping of long routings by examining qualified patterns only. On the other hand, vertex shader codes generally exhibit more demands on central register file accesses. Uniforms of vertex shader codes are stored as live-ins into the central register file. The base addresses of various attributes are also stored in the central RF. Thus, the contention among the operations that access the central RF is very severe. To limit the routing pattern for fast scheduling, **FMS** explores only two routing patterns: one is the direct path and the other is an indirect path via another FU as a routing resource as described in Section 3. Whenever either of these two routing lengths is not sufficient to resolve conflicts on read ports of the central RF, the II should be prolonged. The length of II is translated into the number of scheduling trials, to which the compile time is proportional. Further, a II is inversely related to the performance of the scheduled loop. **EMS**, however, exploring more routing options

Fig. 14. Performance comparison to **EMS** for pixel shader codes.

Fig. 15. Performance comparison to **EMS** for vertex shader codes.

Fig. 16. Comparison of register usage against EMS.

than **FMS**, may successfully find a routing solution without increasing II. We are currently investigating solutions for this problem, like inserting a mov operation before a load/store operation. Overall, the compile time and performance of **FMS** for pixel shader codes are superior to that for vertex shader codes.

5.5. Register Usages

To see how the frequent utilization of indirect paths through LRF by **FMS** affects the total register usage, we compared the measured physical register usage from **FMS** and **VFMS** to that from **EMS**. The result is plotted in Figure 16. Though **FMS** uses more registers than **EMS** for some of the benchmarks such as P12 and V1 through V4, it does not spend more registers on average than **EMS**. In shader codes P12 and V1 through V4, there are very long dependence edges, for which many physical registers are inevitable for **FMS**. **EMS** can avoid using too many registers on these edges by routing data through various components, not just one LRF. Especially, physical register usage in **VFMS** is smaller than the usage of **FMS** since **FMS** gives more priority to exploit a direct path rather than an indirect path if possible, and **VFMS** provides more room for a direct routing path due to doubled II. In consequence, the side effect of **FMS** on register allocation is relatively negligible.

6. RELATED WORK

Many CGRA-like architectures have been proposed with different characteristics in performance, scalability, and programmability. RaPiD [Ebeling et al. 1997] consists of ALUs and memories in a single-dimensional layout, and the network connection between the elements is reconfigurable. Morphosys [Lu et al. 1999] is an 8×8 grid processing unit with a complex network. PipeRench [Goldstein et al. 1999] focuses on pipelining and adopts a structure where the functional units are arranged in stripes.

ADRES architecture [Mei et al. 2003] proposes an architecture template, composed of functional units with a mesh-style network and central and decentralized registers.

There also have been many research efforts on compilers that exploit ILP on CGRAs. It is well known that modulo scheduling problems for unified VLIW architecture are an NP-hard problem [Codina et al. 2002]. The same problem for clustered VLIW is known to be a combination of two NP-hard problems. CGRA modulo scheduling is a set of tightly coupled NP-hard problems. Due to modulo constraints and asymmetric routing resources, it is almost impossible to separate placement and routing problems from each other, unlike the FPGA P&R problem [Mei et al. 2002].

Mei et al. [2002] apply iterative modulo scheduling algorithm on their DRESC framework. Random placements followed by the explicit routing stage are systematically taken by the simulated annealing technique. The DRESC compiler framework adopts a cost function similar to FPGA P&R's PathFinder algorithm [McMurchie and Ebeling 1995; Ebeling et al. 1995]. DRESC starts its scheduling attempt from the theoretical minimum initiation interval (II) and increases II subsequently until it succeeds. For each II, it first generates an initial schedule, which respects dependency constraints, but may overuse resources. Then it makes random variations from the initial schedule by random relocations of operations and rerouting. Each variation is estimated by a cost function, which is constructed by taking account of the overused resources and the penalty associated with them. Note that the route between the producers of the newly placed operation and the operation has to be rebuilt. It may also lead to additional rerouting of other connected operations. Through those repetitive trials and errors, DRESC produces high-quality scheduling results but involves a huge computational cost. As the problem size grows, the whole compilation time of DRESC is unacceptable due to interdependency among scheduling, placement, and routing.

Edge-centric Modulo Scheduling (EMS) [Park et al. 2008] gives priority to routing over placement with an observation that routing is a key step in CGRA modulo scheduling. It suffers a slight degradation in resulting performance but saves compilation time considerably compared to DRESC. Oh et al. [2009] and Kim et al. [2012] mainly focus on efficient handling of recurrences occurring in the EMS approach. Recently, Chen and Mitra [2012] formulated CGRA scheduling into a graph mapping problem. Note that all those techniques have explicit routing stages, which are search based. The proposed technique is distinct from those techniques in that its routing algorithm is effectively folded into the placement algorithm by confining routing paths to prequalified ones. Hence, the routing stage is extremely simplified so that the proposed algorithm enables dramatically fast scheduling even usable for real-time shader compilation.

There are many works related to routings in FPGA's P&R. Our work shares a similarity to HARP (Hard-wired Routing Pattern FPGAs) in that predefined routing patterns are utilized [Sivaswamy et al. 2005]. Recently, innovative modeling and methods have been proposed for PCB escape routings [Yan and Wong 2009]. Although these are excellent routing algorithms, it is not clear how they can be directly used in CGRA modulo scheduling to result in loosening the coupling of routing and placement problems as the routing through our qualified patterns do.

There is no clear architectural difference between CGRA and clustered VLIW [Mei 2005]. Clustered VLIWs and their intercluster communications are well categorized in Fisher et al. [2005] and Terechko et al. [2003]. In addition, much work has been done toward compiling for clustered VLIW [Ellis 1985; Özer et al. 1998; Nystrom and Eichenberger 1998; Sánchez and González 2000]. Some of the concepts from these works can be adapted for CGRAs, but they generally do not consider the issue of routing values through sparse interconnect, except for implicit or explicit data copies between clusters. On the other hand, CGRAs often have more sparsely connected

networks, where many routing resources are shared. The proposed algorithm extracts out the useful routing patterns from those architectures and enables fast compilation by exploiting them.

7. CONCLUSION

This article proposes a fast modulo scheduling algorithm, which improves the compilation speed by multiple orders compared to EMS, the state-of-the-art CGRA modulo scheduler. To reduce compilation time, the proposed scheduler simplifies the routing stage of modulo scheduling so that it can be effectively integrated into the placement stage. By utilizing qualified routing patterns, the scheduler can determine a placement guaranteed not to be invalidated by further routings. We examined the proposed scheduler with shaders of a real-world benchmark in the 3D graphics domain, which is known to be extremely intolerant to long compilation times. In the benchmark results, the proposed scheduler achieves 310 times on average and up to 6,000 times improvement in compilation speed over EMS. The performance degradations of the resulting schedules remain within 30% on average.

Our future research direction includes improving the performance by inserting explicit move operations into a DFG to restore the performance degradation from central RF port contention.

ACKNOWLEDGMENTS

The authors thank our helpful supervisors, Donghoon Yoo, Soojung Ryu, and VP. Jay Kim, for supporting this research effort. We also thank our generous supervisors, Seungwon Lee, Do-Hyung Kim, and VP. Shihwa Lee, for encouraging us to finish up this work. Finally, we thank the anonymous reviewers for their constructive comments.

REFERENCES

- CHEN, L. AND MITRA, T. 2012. Graph minor approach for application mapping on cgras. In *Proceedings of 2012 International Conference on Field-Programmable Technology (FPT'12)*. IEEE, 285–292.
- CODINA, J. M., LLOSA, J., AND GONZÁLEZ, A. 2002. A comparative study of modulo scheduling techniques. In *Proceedings of the 16th International Conference on Supercomputing (ICS'03)*. ACM, New York, NY, 97–106.
- EBELING, C., CRONQUIST, D. C., FRANKLIN, P., SECOSKY, J., AND BERG, S. G. 1997. Mapping applications to the rapid configurable architecture. In *Proceedings of 5th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'97)*. IEEE Computer Society, 106–115.
- EBELING, C., MCMURCHIE, L., HAUCK, S. A., AND BURNS, S. 1995. Placement and routing tools for the triptych fpga. *IEEE Transactions on Very Large Scale Integration Systems* 3, 4, 473–482.
- ELLIS, J. R. 1985. *Bulldog: A compiler for VLIW Architectures (Parallel Computing, Reduced-Instruction-Set, Trace Scheduling, Scientific)*. Ph.D. thesis, Yale University.
- FISHER, J. A., FARABOSCHI, P., AND YOUNG, C. 2005. *Embedded Computing—A VLIW Approach to Architecture, Compilers, and Tools*. Morgan Kaufmann.
- GOLDSTEIN, S. C., SCHMIT, H., MOE, M., BUDIU, M., CADAMBI, S., TAYLOR, R. R., AND LAUFER, R. 1999. Piperench: A coprocessor for streaming multimedia acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*. 28–39.
- KIM, W., YOO, D., PARK, H., AND AHN, M. 2012. SCC based modulo scheduling for coarse-grained reconfigurable processors. In *Proceedings of 2012 International Conference on Field-Programmable Technology (FPT'12)*. IEEE, 321–328.
- LALONDE, P. AND SCHENK, E. 2002. Shader-driven compilation of rendering assets. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'02)*. ACM, New York, NY, 713–720.
- LU, G., SINGH, H., LEE, M.-H., BAGHERZADEH, N., KURDAHI, F. J., AND FILHO, E. M. C. 1999. The morphosys parallel reconfigurable system. In *Proceedings of the 5th International Euro-Par Conference on Parallel Processing (Euro-Par'99)*. Vol. 1685. Springer, 727–734.

- McMURCHIE, L. AND EBELING, C. 1995. Pathfinder: a negotiation-based performance-driven router for fpgas. In *Proceedings of the 3rd International ACM Symposium on Field-Programmable Gate Arrays (FPGA'95)*. 111–117.
- MEI, B. 2005. *A Coarse-Grained Reconfigurable Architecture Template and Its Compilation Techniques*. Ph.D. thesis, Katholieke Universiteit Leuven.
- MEI, B., VERNALDE, S., VERKEST, D., MAN, H. D., AND LAUWEREINS, R. 2002. DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. In *Proceedings of the 2002 IEEE International Conference on Field-Programmable Technology (FPT'02)*. 166–173.
- MEI, B., VERNALDE, S., VERKEST, D., MAN, H. D., AND LAUWEREINS, R. 2003. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proceedings of 2003 Design, Automation and Test in Europe Conference and Exposition (DATE'03)*. IEEE Computer Society, 10296–10301.
- NYSTROM, E. AND EICHENBERGER, A. E. 1998. Effective cluster assignment for modulo scheduling. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'98)*. 103–114.
- OH, T., EGGER, B., PARK, H., AND MAHLKE, S. A. 2009. Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'09)*. 21–30.
- ÖZER, E., BANERJIA, S., AND CONTE, T. M. 1998. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'98)*. 308–315.
- PARK, H., FAN, K., MAHLKE, S. A., OH, T., KIM, H., AND SEOK KIM, H. 2008. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th International Conference on Parallel Architecture and Compilation Techniques (PACT'08)*. 166–176.
- PEEPER, C. AND MITCHELL, J. L. 2003. Introduction to the directX 9 high level shading language. In *ShaderX2: Introduction and Tutorials with DirectX 9*, W. Engel, Ed. Wordware, Plano, Texas.
- RAU, B. R. 1994. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO'94)*. 63–74.
- RIGHTWARE. 2011. Basemark ES 2.0. In <http://www.rightware.com/benchmarking-software/product-catalog>.
- ROST, R. J., LICEA-KANE, B., GINSBURG, D., KESSENICH, J. M., LICHTENBELT, B., MALAN, H., AND WEIBLEN, M. 2009. *OpenGL Shading Language*, 3rd ed. Addison-Wesley Professional.
- SÁNCHEZ, J. AND GONZÁLEZ, A. 2000. Modulo scheduling for a fully-distributed clustered VLIW architecture. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'00)*. 124–133.
- SIVASWAMY, S., WANG, G., ABABEI, C., BAZARGAN, K., KASTNER, R., AND BOZORGZADEH, E. 2005. Harp: Hard-wired routing pattern FPGAs. In *Proceedings of the ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays (FPGA'05)*. ACM, 21–29.
- TERECHKO, A., THENAFF, E. L., GARG, M., VAN ELJNDHOVEN, J. T. J., AND CORPORAAL, H. 2003. Inter-cluster communication models for clustered vliw processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)*. IEEE Computer Society, 354–364.
- YAN, T. AND WONG, M. D. F. 2009. A correct network flow model for escape routing. In *Proceedings of the 46th Design Automation Conference (DAC'09)*. ACM, 332–335.

Received June 2013; revised November 2013; accepted November 2013