Evaluator-Executor Transformation for Efficient Pipelining of Loops with Conditionals

YEONGHUN JEONG, LG Electronics*
SEONGSEOK SEO and JONGEUN LEE, UNIST

Control divergence poses many problems in parallelizing loops. While predicated execution is commonly used to convert control dependence into data dependence, it often incurs high overhead because it allocates resources equally for both branches of a conditional statement regardless of their execution frequencies. For those loops with *unbalanced* conditionals, we propose a software transformation that divides a loop into two or three smaller loops so that the condition is evaluated only in the first loop, while the less frequent branch is executed in the second loop in a way that is much more efficient than in the original loop. To reduce the overhead of extra data transfer caused by the loop fission, we also present a hardware extension for a class of Coarse-Grained Reconfigurable Architectures (CGRAs). Our experiments using MiBench and computer vision benchmarks on a CGRA demonstrate that our techniques can improve the performance of loops over predicated execution by up to 65% (37.5%, on average), when the hardware extension is enabled. Without any hardware modification, our software-only version can improve performance by up to 64% (33%, on average), while simultaneously reducing the energy consumption of the entire CGRA including configuration and data memory by 22%, on average.

Categories and Subject Descriptors: C.1.3 [Other Architecture Styles]: Heterogeneous (hybrid) systems; D.3.3 [Language Constructs and Features]: Control structures; D.3.4 [Processors]: Compilers

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Coarse-grained reconfigurable architecture (CGRA), conditional statements, control divergence, software pipelining, loop fission, predicated execution

ACM Reference Format:

Jeong, Y., Seo, S., and Lee, J. 2013. Evaluator-executor transformation for efficient pipelining of loops with conditionals. *ACM Trans. Architect. Code Optim.* 10, 4, Article 62 (December 2013), 23 pages.
DOI: <http://dx.doi.org/10.1145/2555289.2555317>

1. INTRODUCTION

The need for higher energy efficiency across all domains of computing from embedded to supercomputers is the main driver behind the increasing popularity of accelerators, which include applications specific hardware as well as more programmable ones such as reconfigurable architectures and GP-GPUs. For those architectures, one important source of energy efficiency is Loop-Level Parallelism (LLP), and pipelining is the

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded in part by the Ministry of Science, ICT and Future Planning (2013R1A1A1005534) and in part by the Ministry of Education, Science and Technology under grant 2010-0011534.

Author's addresses: Y. Jeong, S. Seo, and J. Lee, School of ECE, Ulsan National Institute of Science and Technology (UNIST), Ulsan, 689-798, Korea.

J. Lee is the corresponding author of this article (e-mail: jlee@unist.ac.kr).

*This work was done while the first author was in UNIST.

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-ART62 \$15.00

DOI: <http://dx.doi.org/10.1145/2555289.2555317>

Fig. 1. Frequency and size of then-blocks in loops from embedded and vision benchmarks. Conditional statements in loops have highly unbalanced branches in terms of both size and execution frequency. Of special interest are the then-blocks in loops of region *B* and the else-blocks in loops of region *D*, which are large but rarely executed.

preferred way of exploiting LLP on reconfigurable architectures, which have abundant, possibly heterogeneous, processing elements. However, except for a few application domains such as Digital Signal Processing (DSP), even loops can be quite complicated, and thus pipelining a loop optimally for a Coarse-Grained Reconfigurable Architecture (CGRA) that supports dynamic reconfiguration is an area of active research [Hamzeh et al. 2012; Han et al. 2010; Chen and Mitra 2012; Kim et al. 2012b; Park et al. 2008].

One important problem in this context is how to handle conditional statements inside loops. Typically, conditional statements are handled using *predicated execution* [Mahlke et al. 1995; Paar et al. 2002]. However, predicated execution on a dynamically reconfigurable CGRA is wasteful because it allocates resources for both branches of a conditional, even if, quite certainly, only one of them is to be executed. To address the problem of wasted resources, a technique called *Dual-Issue Single-Execution* (DISE) [Han et al. 2010] was proposed for a CGRA. The key idea is to issue operations from both paths simultaneously, but at runtime, execute only one of them depending on the predicate value. This can save cycles and energy, but unfortunately becomes less effective when the sizes of the branches are unbalanced, which is often the case in applications.

We have examined conditionals¹ inside loops from applications in MiBench [Guthaus et al. 2001] and a computer vision benchmark suite [Venkata et al. 2009], in terms of both the relative size and execution frequency of the then- and else-blocks. Our findings are summarized in Figure 1, where the x-axis represents the size balance and the y-axis the execution frequency balance. The center of the graph thus represents the perfect balance whereas the right (left) side is where then-blocks (else-blocks) are larger, and the top (bottom) side is where then-blocks (else-blocks) are more frequently executed.

¹These do not include loop control branch instructions but only those explicitly stated in the source code.

This graph² reveals several useful insights. First, the majority (78%, or 25 out of 32) of the conditionals inside loops have then-blocks only (no else-blocks). Even the rest of them are mostly unbalanced, leaving only two loops near the 50% line in terms of size. This suggests that most loops at least in the applications we have examined have quite unbalanced conditionals, rendering techniques like DISE unattractive. Second, from the vertical distribution, we see that most conditionals are very polarized, strongly favoring one branch over the other. This has an important implication that while predicated execution is very resource-efficient for the if-then statements in region *A* because there then-blocks are almost always executed, for those in region *B*, it is very wasteful, presenting an opportunity for improvement over predicated execution. Third, interestingly enough, much more loops are found in region *B* than in region *A*, pointing to the need for addressing the problem. Thus, our goal in this article is to develop a technique that improves the efficiency of loops belonging to regions *B* and also *D* due to the symmetry of the axes. Our technique is complementary to predicated execution, which is already efficient for loops in regions *A* and *C*.

In this article, we propose a compiler technique that splits a loop with one or more conditionals into multiple smaller loops to achieve more efficient execution on CGRAs. In the simplest case,³ we divide a loop into two loops, where the first one only evaluates the condition while the second one executes the then-block for those iterations where the condition is true. It is very similar to loop fission, but we *compress* the second loop for more efficient execution. We call this *evaluator-executor transformation*, and its performance and energy improvement comes from the fact that it can bypass unnecessary iterations completely, not wasting any energy or resources for them. There are some challenges however. First, while splitting a loop is rather straightforward for those loops with only one if-then statement, it is not obvious how to handle if-then-else or other more complex conditionals including nested and/or a series of conditionals. Second, splitting a loop in this way can incur some overhead, due to the need to pass some variables from one loop to another, which needs to be minimized. Lastly, loops with inter-iteration data dependence (recurrent loops) may not be easily fissioned. We present two techniques—one is a pure software technique and the other is based on hardware extension—to address those challenges.

Our experiments using loops with conditionals from MiBench and computer vision benchmarks demonstrate that our hardware technique can improve the performance of loops with conditionals over predicated execution, by up to 65% (37.5%, on average), across applications. Our software-only version, which does not require any hardware modification, can improve the performance over predicated execution by up to 64% (33%, on average). While our software transformation generates loops that use more memory operations than the original, thereby increasing the dynamic energy in the memory, the increase is very modest, and in many cases much less than what we save on the static energy through runtime reduction. Overall, our software technique can reduce the energy consumption of the entire CGRA including configuration and data memory by 22%, on average.

²It is worth mentioning that those 32 loops (i.e., the potential target of our technique) account for a very significant portion of the application runtime. Our detailed simulation using the SimpleScalar simulator [Austin et al. 2002] and DRAMsim [Wang et al. 2005] modeling a typical embedded system with a dual-issue, in-order processor running at 720MHz, two-levels of cache hierarchy, and a DDR-333 SDRAM, indicates that the target kernels which are located in the high opportunity regions account for 41%, on average, of the total application runtime (arithmetic mean), while they are only one-third of all the important loops (see Table I). We believe that it is because larger loops tend to be more complicated and therefore more likely to contain conditional statements that we find a disproportionately larger portion of the application runtime spent in our target loops.

³More generally, our technique can handle loops with multiple if-then-else conditionals inside.

Fig. 2. Our target CGRA architecture.

2. TARGET ARCHITECTURE

Our target architecture is a Coarse-Grained Reconfigurable Architecture (CGRA) that is dynamically reconfigurable. A CGRA [Hartenstein 2001] is a type of reconfigurable architecture where each Processing Element (PE) operates at the granularity of words rather than bits as in FPGAs. This has both advantages and disadvantages. The advantages include fast runtime reconfiguration, high efficiency for operations directly supported by the PE such as arithmetic operations, and lower programming barrier for software engineers with no hardware design background. The disadvantages include inefficiency in bit-level manipulation or in general any operation that is not directly supported by the “instruction set” of a PE. Also, the lack of flexibility in terms of trading off the cycle time for routability is another difference from FPGAs.

Our target CGRA mainly consists of a PE array, which is a 4×4 array of heterogeneous PEs, and a multi-bank local memory (see Figure 2). Arithmetic and logic operations can be performed by any PE, but expensive operations such as multiplication can be performed by some PEs only. There are also specific PEs that can perform memory operations, called *load-store PEs*. The number of load-store PEs is matched with the number of banks in the local memory. The local memory is a scratch pad memory, which provides guaranteed access time with no memory stall. Thus, it is the compiler’s responsibility to ensure that the data accessed by load-store PEs are present in the CGRA’s local memory. To provide flexibility in accessing different banks from any load-store PE, there is a crossbar switch between the load-store PEs and the memory ports.

The PE array may employ any interconnect topology among the PEs such as simple mesh or mesh plus diagonal. The interconnect architecture connects the output port of one PE to its neighboring PEs’ input ports. Each PE’s output is registered, and the PEs all operate in lock-step. Hence, pipelined execution is a natural way to exploit the parallelism of a CGRA.

The instruction of a PE is called *configuration*, which can be quickly changed to another thanks to the distributed configuration cache. While this fast reconfiguration mechanism greatly extends the capability of a limited-sized CGRA for large tasks, general branch operations are still not supported on most CGRA architectures, posing a challenge for application mapping.

3. RELATED WORK

There are various ways of mapping loops onto CGRAs [Cardoso and Diniz 2009; Lee et al. 2003], but loop pipelining, or software pipelining-based mappings [Mei et al.

2003b; Park et al. 2008] can effectively utilize heterogeneous PEs yielding high-throughput schedules and can also handle recurrent loops. On the other hand, for parallel loops on homogeneous PEs, Single-Instruction Multiple-Data (SIMD) mappings [Kim et al. 2012a] can lead to higher utilization because, unlike in software pipelining, no PE needs to be wasted for *routing*.

Control divergence poses many challenges when mapping loops for CGRAs. Software pipelining-based mappings typically use predicated execution [Rau 1994], which converts control flow into dataflow, by issuing instructions from both branches of a conditional (thus, no control dependence) but executing them only if the predicate is true (thus data dependence). Predicated execution requires hardware support such as (i) adding predicated versions to some (*partial predication*) or all (*full predication*) instructions; (ii) a predicate register file, which may be shared among all PEs; and (iii) one or more predicate-defining instructions that can write to the predicate register file. Our work assumes that the CGRA supports predicated execution for conditional execution.

Recent work for CGRAs tries to improve the efficiency of control flow in loops. Dual-Issue Single-Execution (DISE) [Han et al. 2010] issues two instructions simultaneously but executes only one of them with the “true” predicate, which can increase performance, especially when the conditional has balanced branches. State-based Full Predication (SFP) [Han et al. 2013], on the other hand, can achieve energy saving by putting a PE into a low power mode if the PE is to receive operations that will be disabled due to predication. Although SFP has been applied to mappings that are not pipelined, SFP for pipelined mappings is not known.

Bundled Execution of REcurring Traces (BERET) [Gupta et al. 2011], which is a hardware accelerator tightly coupled to the CPU pipeline, also exploits the existence of a *common scenario* in a control flow. However, instead of fissioning loops, BERET restarts an iteration on the main processor whenever control diverges from the common scenario, which is monitored by the BERET hardware. Unlike BERET, CGRA has a large overhead for switching to the main processor, making it very inefficient if BERET were applied to CGRAs. Also, whereas BERET does not software-pipeline loops, which makes it easy to restart an iteration, our proposed technique apply software pipelining to each loop after evaluator-executor transformation.

GPUs also suffer from control divergence in loops, which can be addressed by dynamically forming SIMD-instructions from large collections of threads [Fung and Aamodt 2011; Rhu and Erez 2012]. Simultaneous Branch and Warp Interweaving [Brunie et al. 2012] executes operations from divergent control paths or even from different scheduling groups (warps) *simultaneously* as in the same SIMD instruction. For unstructured control flows, finding out when divergent threads reconverge can be important and challenging, which is addressed by Damos et al. [2011].

In high-level synthesis, previous work handles conditional statements in pipelined loops using techniques like path-based scheduling algorithms [Camposano 1991], speculation techniques [Gupta et al. 2001], or static branch prediction [Holtmann and Ernst 1995].

Our work is based on loop fission [Aho et al. 2006], a well-known loop transformation technique. Our technique specializes loop fission for handling control divergence and extends it by compressing the second loop, which is the main source of performance and energy improvement. A similar loop transformation called *inspector-executor strategy* was proposed [Saltz et al. 1997] to handle data-dependent memory access patterns. It fissions a loop into two, with the first calculating the addresses of the data to be fetched/stored while the second is using the information to do the actual computation.

4. EVALUATOR-EXECUTOR TRANSFORMATION

4.1. Basic Idea

The basic idea of our evaluator-executor transformation is to divide a loop into smaller ones, such as one that includes operations before and up to the condition evaluation, and one that includes operations after the condition evaluation. The evaluator loop also stores the iterator values for the iterations where the condition holds true, and the iterator values are later retrieved in the executor loop. Improvement of performance and/or energy is the main motivation, as the executor loop often has a much lower trip count than the original loop. We also extend this basic idea to loops that have if-then-else conditionals and those with more than one conditionals.

4.1.1. Illustrative Example. Figure 3 illustrates a simple loop with an if-then statement. The control flow graph of the loop body has only two basic blocks. The condition expression is evaluated in the first block, and only when the condition is true, the second basic block is executed.

In the conventional mapping based on predicated execution only, the control dependence is converted into data dependence, creating data dependence edges from 5 to 6 ~ 9 in Figure 3(b). These additional edges may seem to complicate routing for mapping, but the hardware architecture may provide special support for routing of predicate values, which are only one-bit signals. The main weakness of predicated execution, however, is the waste of resources allocated to the predicated block (the second block in our example), which really needs to be executed only when the condition is true. But since the condition value changes from iteration to iteration and the loop schedule must be generated statically, resources must be allocated for the second block regardless of the condition value, which leads to waste of resources.

Instead, we create two loops, namely *evaluator* loop and *executor* loop. Essentially, the evaluator loop contains the code up to the condition evaluation, and the executor loop contains the code below the condition expression. To preserve the functionality, a few operations need to be added, such as saving and recalling of the iterator values for true iterations.

4.1.2. Challenges and Problems. While the transformation may seem trivial for the previous example, as mentioned before, the control structure in the loop body in general may be much more complicated. Also if there is a statement after the conditional, simple two-loop fission will not be enough. Thus, the first challenge is, given an arbitrary control flow graph of a loop body, to determine how to partition the basic blocks to fissioned loops so that the functionality is preserved and the benefit of the loop fission is maximized. Recurrent loops, or loops with inter-iteration data dependence, pose another challenge, as data dependence cycles should not be broken during loop fission. In some cases, we can circumvent data dependence cycles and still perform our loop transformation. Another challenge is intra-iteration data dependence such as a scalar variable defined before, but used inside, a conditional statement. Such variables need to be passed to the executor loop, which requires adding new operations to evaluator and/or executor loops. This is a by-product of loop fission and since it can offset the performance and energy advantage of our technique, this overhead needs to be minimized.

4.2. Overall Flow

Figure 4 lists the overall flow of our evaluator-executor transformation. Given a loop, its Program Dependence Graph (PDG) [Ferrante et al. 1987], which is an intermediate representation that makes explicit both the control and data dependences for each operation, is partitioned into three sets, which we call simply *A*, *B*, and *C*, where

Fig. 3. Splitting a loop with an if-then statement into evaluator and executor loops. (a) Original loop, (b) its program dependence graph (PDG), (c) evaluator and executor loops, and (d) their PDGs. Circled numbers in (c) correspond to the steps in our flow in Figure 4.

①	PDG partitioning
②	Inter-iteration data dependence handling
③	PDG modification
④	Intra-iteration data dependence handling
⑤	PDG optimization
⑥	PDG scheduling and mapping
⑦	Comparison with the original loop

Fig. 4. The overall flow of our evaluator-executor transformation.

Fig. 5. Various control flow graphs.

set C may be empty. These sets correspond to *evaluator*, *executor*, and *trail* loops, respectively. Next, inter-iteration data dependence analysis is performed to see if there is recurrence cycle(s) that cannot be handled. Some loops with recurrence cycles can be successfully fissioned; for others, our transformation fails. Then the PDG of each set is modified by adding operations necessary for our transformation. The next step is intra-iteration data dependence handling, which identifies the variables that need to be passed from the evaluator to the executor or the trail loop, but which do not create any dependence cycle. Such “temporary” variables may be passed through the local memory or simply recomputed—we decide it by comparing their estimated cost. The next step, PDG optimization, cleans the PDG for any possible dead code introduced during the previous steps (e.g., recomputing a temporary variable in the executor loop may make the original definition in the evaluator loop unnecessary). Finally, the PDG of each loop is scheduled and mapped onto the target architecture, and the expected runtime is compared with that of mapping the original loop using predicated execution only, with the better being selected.

We now explain the key steps of the flow, starting with the PDG partitioning.

4.3. PDG Partitioning

The problem of PDG partitioning is to determine which basic blocks to include in each fissioned loop, which is far from trivial for an arbitrary control flow graph such as the one in Figure 5(c). This problem arises even for a simple if-then-else statement because the executor loop cannot contain both the then- and else-blocks, which defeats the purpose of our transformation. Another challenge is that at this stage, we simply cannot foresee all the effects of choosing one block over the other, due to the intricacies caused by later steps such as inter- and intra-iteration data dependence handling and PDG scheduling and mapping. Thus, we develop an intuitive metric for the usefulness of a block as (part of) the executor loop, which is generalized and used in our partitioning heuristic for PDGs.

Before we proceed, let us clarify one underlying assumption here. To efficiently support an arbitrary control flow graph, or even a simple if-then-else statement in a loop, we assume that the architecture supports predicated execution. Otherwise, we have to create a loop for each of the then- and else-blocks, which incurs much more overhead and is likely to cancel out any performance advantage of our transformation. With this architecture, control statements are allowed in any of the fissioned loops.

In the following, we first develop an intuition for the usefulness metric, consider the trail loop case, and present our algorithm.

4.3.1. Usefulness Metric. To explain the intuition behind our usefulness metric for including a block in the executor loop, let us consider a simple loop that contains only one if-then-else statement without any statement afterward, as illustrated in Figure 5(a). Let us assume that the branch probability is p , and the then- and else-blocks take S_T and S_E cycles, on average, respectively (which may be approximated to the number of operations in each block). Further, the loop iterates N_{orig} times, and the condition block takes S_C cycles, on average.

In this example, there are only two choices for the executor loop: the then-block or the else-block. Depending on which block is included in the executor loop, the expected runtime after loop fission may vary. Assuming that the runtime of two blocks executing together is the sum of the runtime of each running separately⁴ and ignoring the control overhead, the total runtime of the fissioned loops in each case can be (approximately) represented as follows:

$$\begin{aligned} T_{with-then-in-executor} &= (S_C + S_E)N_{orig} + S_T N_{orig} \cdot p \\ T_{with-else-in-executor} &= (S_C + S_T)N_{orig} + S_E N_{orig}(1 - p). \end{aligned} \quad (1)$$

Note that the aforementioned formulas include the effect of both then- and else-blocks, and that when the then-block is chosen for the executor, the else-block is assumed to be part of the evaluator (using predicated execution) and vice versa, which explains the term $S_E N_{orig}$ in the first line and $S_T N_{orig}$ in the second. For the then-block to be the better choice, we require $T_{with-then-in-executor} < T_{with-else-in-executor}$, or $\frac{S_T}{p} > \frac{S_E}{1-p}$, which means that the best candidate for the executor loop is a large and rarely executed one. In other words, the usefulness of a block for the executor loop is proportional to the block size divided by the execution frequency. We extend this metric to any set of blocks to define *usefulness metric* of a set of blocks as their total size divided by their execution frequency. This metric is used in our partitioning algorithm, which first identifies all the candidate sets of blocks, and chooses the one that maximizes the usefulness metric for the executor loop.

4.3.2. Trail Loop. A slightly more complex case is one that has statements after a conditional, as illustrated in Figure 5(b). Such statements may have data or control dependence on one or more statements in the conditional. In particular, if the post-conditional statements have data or control dependence on the blocks included in the executor loop, they must be executed after the executor loop as a separate loop, which is called *trail loop*.

In a nested control statement, determining which blocks should be included in the trail loop may not be obvious. For instance, in Figure 6 if the executor includes block ❶ only, the trail loop does not include blocks ❷, ❸, or ❹, but block ❺ only. The principle here is that all the blocks that may be executed in an iteration where the executor block is executed must be included in the trail loop if they are executed after the executor

⁴This assumption holds true for an in-order RISC processor with a single pipeline. While our CGRA is an in-order processor, it has multiple pipelines, which can decrease the accuracy of our performance model.

Fig. 6. Determining the set of blocks for the trail loop.

block. Based on this observation, we provide an algorithmic solution for the general case in the next section.

4.3.3. General Solution. Our solution for an arbitrary Control Flow Graph (CFG) generated from a structured code (i.e., without gotos) is as follows: (i) identify all the candidate sets of blocks, (ii) choose for the executor the one that maximizes our usefulness metric, that is, size over execution frequency, (iii) select the blocks for the trail loop, and (iv) the remaining blocks become the evaluator. For this algorithm, we assume that the branch probabilities of the conditionals are available through profiling or by other means (e.g., static analysis). In the following discussion, we assume without loss of generality that every conditional statement has up to two cases—if-then or if-then-else statements only.

First, we associate a unique predicate variable with each condition expression as illustrated in Figure 5(c). Then we can find the predicate expression that dictates the condition for executing each block in the CFG, which is a product of predicate variables or their complements (called *literals*), as shown inside each block in the figure. Let us call this product of literals *execution condition* of the block. It is easy to see that when the predicate variable takes the value of the corresponding branch probability, the execution condition of a block becomes the execution frequency of the block. We say that execution condition x *implies* execution condition y if all the literals of y are found in x . For instance, $p_1 p_2 p_3$ implies $p_1 p_2$ but not $p_1 p_2 \overline{p_3}$. Execution condition of a set of blocks is defined as the Boolean OR of each block's execution condition.

Let B denote the set of all the blocks included in the executor loop. Then we note that if B includes a block whose execution condition is x , adding to B any other block whose execution condition implies x can only increase the benefit of the executor, as per our usefulness metric. This is because adding it does not increase the execution frequency of B but only increases its size. This is in effect saying that in a nested conditional statement, all the inner conditionals should be included in their entirety if the outer one is included. In our example of Figure 5(c), these are the candidate sets of blocks (each set is represented by its execution condition): $\{p_1, p_1 p_2, p_1 p_2 p_3, p_1 p_2 \overline{p_3}, \overline{p_1}, \overline{p_1} p_4, \overline{p_1} \cdot \overline{p_4}, \overline{p_1} \cdot \overline{p_5}\}$. Finding all the candidates can be done efficiently by exploiting the nesting property of the conditionals. Once all the candidate sets of blocks are identified, they are evaluated in terms of the usefulness metric, and the one with the highest metric is chosen.

Fig. 7. Recurrent loops can be fissioned if the recurrence can be isolated into one fissioned loop. This example illustrates that by moving the operation in the gray box from the executor to the evaluator, we can break the dependence of the evaluator on the executor.

Then, the set of blocks included in the trail loop, called C , includes all the blocks that (i) appear after the blocks of B in the original code, and (ii) whose execution condition y is *not* exclusive with that of B , that is, $y \cdot x \neq 0$, where x is the union of all the execution conditions of the blocks in B . Figure 5(c) shows set C for the B selected as in the figure.

Finally, all the remaining blocks are automatically included in set A , which is for the evaluator loop. In a later step (e.g., intra-iteration data dependence handling), if the operations in C are found to have no data dependence at all on B , C can be merged into A .

4.4. Inter-Iteration Data Dependence

In a loop, an operation that is dependent on the same or another operation of a previous iteration, either through a scalar or an array variable, can create a cycle in the data dependence graph. Since data dependence represents precedence relationship, such cycles may not be broken into different loops, which can complicate loop fission. One caveat, however, is that a data dependence cycle that remains completely in one of the fissioned loops does not interfere with loop fission. Therefore, we move operations between the sets in order to make a dependence cycle completely contained in one set. Figure 7 illustrates this for a loop with a max operation. As shown in Figure 7(a), the original loop has only one if-statement. Therefore, the obvious choice for set B is the

```

1: identify all inter-iteration dependence cycles
2: for each dependence cycle,  $DC$  do
3:   find the blocks to which operations of  $DC$  belong
4:   if  $DC$  consists of operations from more than one blocks then
5:     if  $DC$  includes any operation from  $A$  then
6:       for each operation  $\alpha \in DC$  that belongs to  $B$  or  $C$  do
7:         move  $\alpha$  and all its predecessors to  $A$ 
8:       end for
9:     else
10:      for each operation  $\alpha \in DC$  that belongs to  $B$  do
11:        move  $\alpha$  and all its successors to  $C$ 
12:      end for
13:    end if
14:  end if
15: end for
16: fail if no operation remains in  $B$ 

```

Fig. 8. Algorithm for handling recurrent loops.

body of the if-statement, whereas the other operations comprise set A including the comparison operation in the condition expression of the if-statement.

However, this partitioning causes a problem in loop fission due to a dependence cycle. The dependence cycle consists of two operations: the max assignment operation in the gray box, which is in set B , and the comparison operation, which is in set A . Since the two operations belong to different sets, it is impossible to create a new loop from each set. However, if we move the max assignment operation from B to A (with necessary predication), both operations comprising the dependence cycle now belong to set A . In other words, the dependence cycle becomes completely contained in A , from which we can create an evaluator loop that is no longer dependent on the executor loop, as shown in Figure 7(b).

Figure 8 shows an algorithm that checks if a loop with inter-iteration data dependence cycles can be fissioned by moving some operations between the sets. In general, moving operations from B to A or C is possible by predicating the operations. On the other hand, it is impossible to move operations from either A or C to B , without duplicating the code. Therefore, we can gather the operations into either A or C . Since they are equivalent in terms of execution frequency, we choose to use A whenever the dependence cycle involves operation(s) from A (line 5). Otherwise, the dependence cycle consists of operations from either B or C only, in which case we gather them to C (lines 9–12). The actual moving of operations in lines 7 and 11 is carried out recursively for all their predecessors or successors depending on the destination set. The predecessor/successor relationship is defined for the entire data dependence graph for all the sets, as opposed to within each set. If all dependence cycles are contained within individual sets without emptying the set B , we can proceed to the next step (line 16). Otherwise, the evaluator-executor transformation is aborted.

4.5. PDG Modification

Our transformation requires the evaluator loop to store two pieces of information: the iterator values for the *true* iterations and the total number of *true* iterations. This can be done by adding a few operations in the PDG of set A , as illustrated in Figure 3. While the evaluator and the trail loops inherit the loop control code (i.e., iterator initialization and increment) of the original loop, the executor loop has a trivial loop control code, and inside the loop body, the real values of the original iterator are retrieved from the

Table I. Applicability of the Proposed Transformation

All important loop	With conditional statement?	Recurrent loop?	Can move recurrence cycle into one sub-loop?	Evaluator-Executor fissionable?
93	Yes (32)	Yes (16)	No (7)	No (7)
		No (16)	Yes (9)	Yes (25)
	No (61)	Not Applicable		

memory. Thus one load operation and related operations (for address calculation) are added to the PDG of set B .

4.6. Intra-Iteration Data Dependence

An example of intra-iteration data dependence is a variable defined in set A and used in B or C . Such variables, which we call *transfer variables*, may not be available in B or C unless they are explicitly copied. One can either save and restore the transfer variable through the local memory after first converting it into an array (similar to *variable expansion* in software pipelining [Rau 1994]), or simply recompute the variable where it is needed. We try both and choose the better one, or the one that requires fewer additional operations, possibly with some weighting factors.⁵ In the case of save-and-restore, a transfer variable used only in B needs to be saved for *true* iterations only. In the case of recomputation, the expression tree corresponding to the definition of the variable needs to be copied from A .

A more complicated case is when a variable is defined in A , modified in B , and used in C . Such variables can be passed through the local memory, which is similar to the simpler case. Recomputing them in C requires copying to C both its definition from A and the conditional update from B , which can increase the overhead of the transformation (thus possibly favoring the save-and-restore method).

4.7. Applicability Study

To see the applicability of our proposed transformation, we study 15 applications⁶ from MiBench [Guthaus et al. 2001] and SD-VBS [Venkata et al. 2009], which is a computer vision benchmark. From those applications, we first collect 93 loops that are the innermost loops in the functions that each accounts for at least 10% of the application runtime. As summarized in Table I, about one-third of them (32 loops) contain one or more conditional statements, or continue or switch statements. They are the potential target of our technique. Half of them (16) are recurrent loops while the other half (16) have no inter-iteration data dependence. In nine of the recurrent loops, the recurrence cycles can be moved into the evaluator loop, enabling our evaluator-executor transformation. Thus the number of applicable kernels is 25 ($= 16 + 9$) out of 32, which suggests a high applicability of our transformation.

5. FISSION-SPECIALIZED CGRA

While the previous section presented our evaluator-executor transformation in a largely target architecture-independent manner, we now propose a hardware extension that is specially tailored for the transformation, for a class of Coarse-Grained Reconfiguration

⁵For example, memory operations may be given higher weights to reflect their higher energy consumption.

⁶Applications are Blowfish decoder, Blowfish encoder, gsm, rsynth, susan corners, and susan smoothing, all from MiBench, in addition to the ones listed in Table II.

Fig. 9. Extending each quadrant (2×2 PE array) of the baseline CGRA with: a FIFO queue, an address generator, Executor Enable (XE) output signals, *pop* operation in one PE, and *push* operation in every PE. A *true* iteration counter is also added to the CGRA. Black-filled components are unique in the entire CGRA. ScratchPad Memory (SPM) is the CGRA's local memory.

Architectures (CGRAs). This architecture extension also accommodates loop fission for CGRAs.

As the extra code in our loop transformation originates from:

- (1) saving and restoring iterator for true iterations,
- (2) counting true iterations, and
- (3) either save-and-restore or recompute transfer variables,

our strategy is (i) to have a hardware counter for counting true iterations and (ii) to provide efficient means of saving and restoring iterator and transfer variables.

5.1. Hardware for True Iteration Counting

Figure 9 illustrates the architecture extension for a quadrant (2×2 PE array) of a baseline 4×4 CGRA, as described in Section 2. The hardware for counting *true* iterations is very simple, consisting of one counter only (True_CNT in the figure), which serves the entire CGRA. The *true* counter is driven by the OR-ed result of all Executor Enable (XE) outputs of Processing Elements (PEs). The XE output of a PE is identical to its predicate output, except that it is driven high only when the PE is actually computing the condition value in an Evaluator loop; thus, usual predicated execution does not interfere with true iteration counting. The CGRA's controller is aware of the *true* counter, and its final value, which is the total number of true iterations, is used as the trip count of the Executor loop.

5.2. Hardware for Save and Restore

5.2.1. Push and Pop. To save and restore the iterator or a transfer variable, a number of operations may be needed, including a store/load operation and the accompanying address calculation computation. What they do, however, is simple push and pop

operations using the CGRA's local memory as a FIFO. Thus, we simplify this by replacing the whole address calculation with a very simple address generator with no multithreading, which can be implemented with a counter, and by embedding the store operation into other instructions as a suboperation controlled by one bit of the configuration.⁷ This suboperation is called *push*, and therefore can be executed by any PE. The load operation is not embedded, but provided as a separate instruction (called *pop*), which is simpler than a load because the address need not be specified.

5.2.2. Optimizing for Performance. To explain the asymmetrical design decision between push and pop, let us consider the runtime of a software-pipelined loop on a CGRA, which can be modeled as $II \cdot (L + N - 1)$, where II , L , and N are the initiation interval, the number of stages (the schedule length in terms of II), and the number of iterations, respectively. Then the runtime difference between the original loop and the fissioned loops is given as follows:⁸

$$\begin{aligned} T_{orig} &= II_{orig}(L_{orig} + N_{orig} - 1) \\ T_{fission} &= II_1(L_1 + N_{orig} - 1) + II_2(L_2 + pN_{orig} - 1) \\ \Delta T &\approx (II_{orig} - II_1 - pII_2)N_{orig}, \end{aligned} \quad (2)$$

where the subscripts, namely, *orig*, 1, and 2, denote the original loop, the evaluator, and the executor, respectively, and p is the executor's execution ratio. The approximation in the last equation is based on the assumption that $N \gg L$ in general. From the last equation, it is clear that to maximize the difference, we must minimize II_1 , or the initiation interval of the evaluator.

Thus, we design *push*, which is used in the evaluator, such that it requires no additional instruction (thus no additional PE) and can be performed on any PE (no pressure on placement or routing). On the other hand, *pop* is designed to require an instruction (one PE needed), and it must be done on a particular PE in each quadrant. Luckily the pressure on routing due to pop is not high, because a pop operation is usually a leaf node of an expression tree and, therefore, has much freedom in terms of placement and routing anyway.

Each quadrant supports one push/pop operation per cycle, and thus a CGRA can access up to four variables including an iterator using push/pop in any loop. If more variables need to be transferred, we do it in software, allocating hardware to the earliest appearing variables.

5.2.3. Queue. Another peculiarity of the push operation is that it may⁹ need to be done only in true iterations, or when the condition expression is true. To complicate matters, there is often a time lapse between the definition of a variable and when the condition value becomes available. Thus, we need to hold the variable temporarily until it can be either committed to the local memory or discarded. While we can hold such a variable temporarily by letting it stay in a PE or keep routing it among PEs until the condition value becomes available, both of them invariably waste resources and add to the routing pressure. Instead, we add a queue, as illustrated in Figure 9, one in each quadrant. A new value is pushed to the queue once in every initiation interval by the CGRA's controller. The decision of whether to commit or discard the data is given by the global version of the XE signal.

⁷This requires reserving one bit per PE in the configuration for the push operation.

⁸For simplicity we assume no trail loop in this analysis, but the effect of one can be lumped into that of the evaluator loop.

⁹Some variables, such as one that is used in the trail loop, may need to be pushed every iteration regardless of the condition value (see Push_Always in Figure 9).

The required length of the queue is determined by how much the two moments, variable definition and condition value availability, are separated in time. In modulo scheduling, the condition value is usually determined at the last stage of the schedule, whereas iterator variables are typically defined at the first stage, which is the worst case. From this worst case, we know that the queue size only needs to be as large as the number of stages in a modulo schedule. Our experimental results using embedded and vision applications indicate that the maximum number of stages is 5.5, on average, across applications, and up to 11 for two applications. Thus, we use a 12-entry queue. Now, this fixed queue size does not mean that certain loops cannot be mapped if they happen to have more stages than the queue size, but it only means that the mapping has to be changed, such that either the number of stages is reduced possibly at the expense of the increased initiation interval, or some variables should wait in PEs during the excess stages.

In summary, our architecture extension is designed to nearly eliminate the code overhead of our evaluator-executor transformation for a modest amount of hardware, as listed in the following:

- (Per PE) adding push operation
- (Per quadrant) adding pop operation to one of the PEs; a FIFO; and a counter for address generation
- (Per CGRA) a counter for counting true iterations.

For a 4×4 CGRA, this hardware overhead is mostly four 12-entry FIFOs and five counters.

5.3. Hardware Synthesis Results

We have implemented the proposed architecture extension in VHDL and Verilog. The key architectural parameters of the baseline CGRA are as given in Section 6.1. Additionally, the configuration register of each PE is 32-bit, and each PE has a private register file with four entries. There is no global or rotating register files, other than the global rotating predicate register file.

Our synthesis results targeting Xilinx FPGA Virtex-5 LX330 indicate that compared to the baseline CGRA, our extended architecture uses 44% more registers and 86% more latches plus four more adders and comparators while the number of slice LUTs increases only marginally by 2.7% from 12,118 to 12,449. The maximum clock speed is almost identical, with 224MHz versus 223MHz (the latter is for the extended architecture).

6. EXPERIMENTS

6.1. Experimental Setup and Applications

We evaluate the effectiveness of our proposed technique on a CGRA that supports predicated execution and consists of a 4×4 heterogeneous PE array and a multi-banked local memory, as described in Section 2. Our CGRA has four load-store PEs, one in each row, that are connected to the four banks of the local memory, where each is a 256KB scratchpad. A load operation from a PE takes six cycles due to a cross-bar switch between the load-store PEs and the memory banks. The interconnection between the PEs includes mesh and diagonal. Every PE can perform ALU operations while more expensive operations (e.g., multiplication) can be done by four specific PEs only. Operations on a PE are fully pipelined and can be predicated. There is no central register file except for a predicate register file, which can be updated by any PE. For mapping, we use our version of modulo scheduling based on the EMS algorithm

Table II. Kernel Characteristics

Application	#Blocks	#Op	f_{ex}	r_{ex}	V
Cjpeg	2	8	4%	25%	1
Dijkstra	3	21	5%	38%	2
Lame_1	2	28	93%	64%	1
Lame_2	3	27	94%	59%	4
Lame_3	3	14	0%	36%	2
Susan_edges	2	216	31%	2%	2
Tiffdither	18	40	6%	13%	4
Tiffmedian_1	9	41	2%	63%	4
Tiffmedian_2	2	18	0%	28%	3
Disparity	2	11	16%	36%	2
Multi_ncut	2	11	0%	45%	2
Stitch	2	14	4%	57%	2

Note. f_{ex} and r_{ex} represent the frequency and the relative size (compared to the entire loop body) of the executor loop, respectively, and V is the number of iterator/transfer variables.

[Park et al. 2008], without the rotating register file support, and with predicated execution support. For our hardware extension, we use a 12-entry queue for each quadrant.

We used 12 loops, which are a subset of the 25 loops in Table I, to which the proposed transformation can be applied, excluding those that cannot be mapped to the CGRA due to the limitations of the CGRA architecture and/or our compiler.¹⁰ Table II lists the characteristics of the loops. The number of operations on Column 3 includes only the ones included in the PDG of the original loop, but not the loop-control related ones, which are taken care of by the CGRA controller. The fourth column shows how often the executor loop is executed per iteration, on average (i.e., execution frequency), and the fifth column is calculated as the ratio of the number of operations in the executor loop to that of operations in the entire loop body, before applying evaluator-executor transformation. The last column is the number of variables including the iterator that need to be transferred between loops.

6.2. Performance of Our Transformation with Hardware Extension

For performance evaluation, we compare three cases in this section. First, we map each loop using predicated execution only, denoted by **P**, which is the baseline of our comparison. Then we apply our evaluator-executor transformation with *hardware extension enabled*, which is denoted by **H**. We also estimate the number of *dynamic* operations that would be executed if the block(s) corresponding to the executor loop were executed only when necessary while the other blocks were executed in every iteration, denoted by **Op** in the graph. We use this formula, $(1 - r_{ex}) \cdot 1 + r_{ex} \cdot f_{ex}$, where r_{ex} and f_{ex} are the size ratio (in the number of operations) of the block(s) selected for the executor loop compared to that of the entire loop body, and their execution frequency, respectively (i.e., the fourth and fifth columns of Table II). This metric represents the fraction of operations that must be executed (over executing all the loop body operations, as is the case with predicated execution, or **P**) in any loop fission-based scheme including ours without considering any overhead of loop fission. Since actual loop fission results will include overheads, the **Op** metric may reveal how effectively our technique can realize the performance potential.

Figure 10 compares the runtime results. First, we note that our scheme with hardware extension follows the number-of-operations prediction very closely. In most cases,

¹⁰The most common reason for mapping failure is too many conditionals inside a loop. Also, loops with identical PDGs are counted as one.

Fig. 10. Comparing runtimes of three cases: P (predicated execution), H (our evaluator-executor transformation with hardware extension), and Op (the number of dynamic operations).

our scheme generates performance that is similar to the number-of-operations prediction, which therefore could be used, for example, in determining whether to apply our transformation technique in a compiler. Surprisingly, however, in several cases our scheme can excel the predictions significantly, including Cjpeg, Dijkstra, and Susan_edges. This is because the mapping result of our scheduler is not exactly proportional to the number of operations of the PDG. Rather our scheduler, which is based on a variant of modulo scheduling [Park et al. 2008] specialized for CGRAs, tends to perform better for smaller graphs because smaller loops usually have lower routing pressure; to wit, scheduling two smaller loops may generate better results than scheduling one large one. The other reason is that in the original loop the iterator has to be routed from the beginning until almost the end of a loop, whereas our transformation often stores the iterator early in the evaluator loop, and therefore does not consume any more routing PEs.

To illustrate the superlinear increase of the runtime in the number of operations, Figure 11 shows the PDGs of Dijkstra loop before and after the transformation. While the number of operations is reduced by 43% only (from 21 to 12), the number of necessary PEs including those used for routing only is reduced by 66% (from 38 to 13). Consequently, the minimum initiation interval on a 4×4 PE array is reduced by 3 times (from 3 to 1).

On the other hand, in the case of Lane_3, the performance of our scheme is much worse than the prediction and is similar to (but still better than) the original. This is a direct consequence of the evaluator, in this particular case, taking almost as many cycles as the original loop—they happen to be scheduled with the same initiation interval, with the evaluator having fewer stages. Thus, although the reduced operation count is very likely to lead to reduced runtime, this may not always be the case, due to the randomness of the scheduling process. Overall, using our hardware scheme can reduce the kernel runtimes by 37.5%, on average, which outperforms the number-of-operations prediction mainly due to the simpler routing.

6.3. Performance of Software Transformation Only

In this section, we compare our evaluator-executor transformation *with* and *without* hardware extension. Figure 12 shows the runtime results, as normalized to that of the predicated execution (P). The rightmost bar (H) represents the case with hardware extension. The other two bars denote software-only cases. Their difference is that for

Fig. 11. Dijkstra loop example, before and after the transformation.

Fig. 12. Comparing runtimes of four cases: P (predicated execution), S1 and S2 (our evaluator-executor transformation without hardware extension), and H (with hardware extension).

transfer variables, **S1** exclusively uses recomputation, whereas **S2** uses the better of recomputation and save-and-restore, based on the number of operations.

From the graph, we observe that while the software-only schemes can improve the runtime significantly compared with the predicated execution, the difference between software-only and hardware extension is only marginal. In particular, our software-only scheme, **S2**, can reduce the kernel runtimes by 33.0%, on average, compared to the predicated execution, which is about 4.5% point difference from that of the hardware version. The same pattern can be seen even in some loops (*Lame_2*, *Tiffmedian*) with the highest numbers of transfer variables. That said, for some other applications, the additional operations added to the software version do affect the performance rather significantly. For instance, *Disparity* and *Stitch*, going from **H** to **S2**, see 30% and 19% runtime increases, respectively.

Between the **S1** and **S2** cases, though the overall difference is small (they are only 3.5% point different in terms of the geometric mean), some applications do see huge runtime reductions. Most notably, the runtime of *Susan_edges* is reduced by 46% from 1.23 times the baseline (when all transfer variables are recomputed) to 0.67 times (when they are all passed via memory), which is because the recomputed expressions in this loop are very large and complex. For those loops, passing transfer variables via memory can eliminate the overhead of recomputation, and generate performance improvement that is comparable to the hardware version.

6.4. Energy Comparison

To evaluate the energy consumption of our proposed scheme, we use the energy model of [Kim et al. 2012b], where the CGRA's dynamic power consists of PE (Processing Element) array power, configuration memory power, and the (CGRA's) local memory power. The PE array power has two components, individual PEs' power dissipation and that of the rest of the PE array. The most interesting parts are the individual PE's power, which is dependent on the operation being performed (ALU, multiply, divide, routing, or idle), and the local memory power, which is proportional to the number of memory accesses. Except for the local memory power, which we obtain from Cacti 6.5 [Thoziyoor et al. 2008] for a 65nm technology, we use the power numbers of [Kim et al. 2012b], scaled to the 65nm technology. The leakage power is assumed to be 50% of the dynamic power with PEs being idle, whereas the leakage power of the local memory is obtained from Cacti. We assume that the CGRA runs at 500MHz.

We compare the total energy consumption of the CGRA and its associated memories in two cases: using predicated execution only (**P**) vs. using our software-only scheme (**S2**). Though faster for many loops, our software-only scheme always generates more memory accesses than the predicated execution. This is because in the predicated execution, load/store operations that are disabled due to predication do not generate memory operations, whereas our scheme generates loops that have extra load/store operations due to the iterator and possibly one or more transfer variables. In both the cases, the underlying hardware is the same.

Figure 13 compares the energy results, overlaid with the normalized runtime results of the **S2** case. The energy consumption is broken down into different categories, among which the leakage accounts for the largest portion, on average. In our energy model, only the local memory energy and the PE array energy are affected by the operations performed on the CGRA. The others are proportional to the runtime, and in those categories we see reductions of about 33%, on average, going from the baseline to our software scheme. The PE array's dynamic energy is only slightly larger than that of the configuration memory, and is overall reduced by our scheme, which is largely a by-product of the reduced runtime. Lastly, the local memory's dynamic energy, which accounts for 22.6% in the baseline, is increased by 16.8% (to 26.4% of the baseline total energy)

Fig. 13. Comparing energy of predicated execution (P) and our software transformed loops (S2). Shown below is the average power dissipation of S2, normalized to that of P.

due to the additional memory operations generated by our software transformation. However, this increase is not excessive in any application. This is because the iterator and optionally transfer variables that are saved in the evaluator loop are *only for those iterations that will be executed in the executor loop*. In other words, we save only the data that will be used later, which helps keep down the energy price we pay in our transformed loops. The additional operations by our technique, coupled with reduced runtime, increases the average power dissipation of our software technique by 16%, on average. Overall, our software technique can reduce the energy consumption of the entire CGRA including configuration and data memory by 22.0%, on average.

6.5. Discussion

While our target architecture shares some features with the ADRES architecture [Mei et al. 2003a] such as the multibanked local memory with a crossbar switch, there are also key differences, which should affect our synthesis results in Section 5.3 as well as performance results. First, our CGRA architecture does not have rotating register files in PEs, but only a private register file in each PE and a global rotating *predicate* register file. The lack of rotating register files results in typically more PEs being used for operand routing, which can help explain the superlinear increase in runtime with larger PDGs. This also means that the performance improvement of our software technique over predicated execution may partially be offset on CGRAs with rotating register files. Second, our CGRA architecture is loosely coupled to the main processor, whereas the ADRES architecture, which can double as a VLIW processor, has a very short control and communication overhead. Since our technique fissions a loop into two or three loops, it has an overhead that is proportional to the amount of the CGRA control latency. Consequently, on a tightly coupled architecture such as ADRES, the scope of our technique could be much higher.

7. CONCLUSION

Many conditionals found in loops have unbalanced branches in terms of both size and execution frequency. Interestingly, often larger branches are less frequently executed, presenting an opportunity for optimization unknown to predicated execution, which blindly allocates resources regardless of the execution frequency. For such loops with

unbalanced conditionals, we presented in this article a software technique that divides a loop into two or three smaller loops so that the condition part is evaluated only in the first loop while the less frequent but larger branch is executed in the second loop in an efficient way. To reduce the overhead of extra data transfer caused by the loop fission, we also present a hardware extension for a class of Coarse-Grained Reconfigurable Architectures (CGRAs). Our experiments using MiBench and computer vision benchmarks on a CGRA demonstrate that our techniques can improve the performance of loops over predicated execution by up to 65% (37.5%, on average), when the hardware extension is used. Without any hardware modification, our software-only version can improve performance by up to 64% (33%, on average), while also reducing the energy consumption of the entire CGRA by 22%, on average.

REFERENCES

- AHO, A. V., LAM, M. S., SETHI, R., AND ULLMAN, J. D. 2006. *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley Longman, Boston, MA.
- AUSTIN, T., LARSON, E., AND ERNST, D. 2002. SimpleScalar: an infrastructure for computer system modeling. *Computer* 35 (2002).
- BRUNIE, N., COLLANGE, S., AND DIAMOS, G. 2012. Simultaneous branch and warp interweaving for sustained GPU performance. *SIGARCH Comput. Archit. News* 40, 3, 49–60.
- CAMPOSANO, R. 1991. Path-based scheduling for synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 10, 1, 85–93.
- CARDOSO, J. M. P. AND DINIZ, P. C. 2009. *Compilation Techniques for Reconfigurable Architectures*. Springer.
- CHEN, L. AND MITRA, T. 2012. Graph minor approach for application mapping on CGRAs. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'12)*. 285–292.
- DIAMOS, G., ASHBAUGH, B., MAIYURAN, S., KERR, A., WU, H., AND YALAMANCHILI, S. 2011. SIMD re-convergence at thread frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44'11)*. ACM, New York, NY, 477–488.
- FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* 9, 3, 319–349. DOI: <http://dx.doi.org/10.1145/24039.24041>.
- FUNG, W. W. L. AND AAMODT, T. M. 2011. Thread block compaction for efficient SIMT control flow. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*. IEEE Computer Society, Washington, DC, 25–36.
- GUPTA, S., FENG, S., ANSARI, A., MAHLKE, S., AND AUGUST, D. 2011. Bundled execution of recurring traces for energy-efficient general purpose processing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44'11)*. ACM, New York, NY, 12–23.
- GUPTA, S., SAVOIU, N., KIM, S., DUTT, N., GUPTA, R., AND NICOLAU, A. 2001. Speculation techniques for high level synthesis of control intensive designs. In *Proceedings of the 38th Annual Design Automation Conference (DAC'01)*. ACM, New York, NY, 269–272.
- GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop (WWC'01)*. IEEE Computer Society, Washington, DC, USA, 3–14. DOI: <http://dx.doi.org/10.1109/WWC.2001.15>.
- HAMZEH, M., SHRIVASTAVA, A., AND VRUDHULA, S. 2012. EPIMap: Using Epimorphism to Map Applications on CGRAs. In *Proceedings of the 49th Design Automation Conference (DAC)*.
- HAN, K., CHOI, K., AND LEE, J. 2013. Compiling control-intensive loops for CGRAs with state-based full predication. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'13)*. EDA Consortium, San Jose, CA, 1579–1582.
- HAN, K., PAK, J. K., AND CHOI, K. 2010. Acceleration of control flow on CGRA using advanced predicated execution. In *Proceedings of the 2010 International Conference on Field-Programmable Technology (FPT'10)*. 429–432.
- HARTENSTEIN, R. 2001. A decade of reconfigurable computing: A visionary retrospective. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'01)*. IEEE Press, 642–649.
- HOLTMANN, U. AND ERNST, R. 1995. Combining MBP-speculative computation and loop pipelining in high-level synthesis. In *Proceedings of the European Design and Test Conference (ED TC'95)*. 550–556.
- KIM, Y., LEE, J., LEE, J., MAI, T. X., HEO, I., AND PAK, Y. 2012a. Exploiting both pipelining and data parallelism with SIMD reconfigurable architecture. In *Proceedings of the International Symposium on Applied Reconfigurable Computing (ARC'12)*, Lecture Notes in Computer Science, Vol. 7199. 40–52.

- KIM, Y., LEE, J., MAI, T. X., AND PAAK, Y. 2012b. Improving performance of nested loops on reconfigurable array processors. *ACM Trans. Archit. Code Optim.* 8, 4, Article 32, 23 pages.
- LEE, J., CHOI, K., AND DUTT, N. D. 2003. Compilation approach for coarse-grained reconfigurable architectures. *IEEE Design Test of Computers* 20, 1, 26–33.
- MAHLKE, S. A., HANK, R. E., MCCORMICK, J. E., AUGUST, D. I., AND HWU, W.-M. W. 1995. A comparison of full and partial predicated execution support for ILP processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. 138–149.
- MEI, B., VERNALDE, S., AND VERKEST, D. 2003a. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Field Programmable Logic and Application*. Number 2778, Lecture Notes in Computer Science. Springer, Berlin, 61–70. Retrieved from http://link.springer.com/chapter/10.1007/978-3-540-45234-8_7.
- MEI, B., VERNALDE, S., VERKEST, D., DE MAN, H., AND LAUWEREINS, R. 2003b. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'03)*, Vol. 1. IEEE Computer Society, Washington, DC.
- PAAR, A., ANIDO, M., AND BAGHERZADEH, N. 2002. A novel predication scheme for a SIMD system-on-chip. In *Proceedings European Conference on Parallel Processing (Euro-Par'02)*, B. Monien and R. Feldmann, Eds., Lecture Notes in Computer Science, Vol. 2400. Springer, Berlin, 834–843.
- PARK, H., FAN, K., AND MAHLKE, S. 2008. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. 166–176.
- RAU, B. R. 1994. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*. 63–74.
- RHU, M. AND EREZ, M. 2012. CAPRI: Prediction of compaction-adequacy for handling control-divergence in GPGPU architectures. *SIGARCH Comput. Archit. News* 40, 3, 61–71.
- SALTZ, J., CHANG, C., EDJLALI, G., HWANG, Y.-S., MOON, B., PONNUSAMY, R., SHARMA, S., SUSSMAN, A., UYSAL, M., AGRAWAL, G., DAS, R., AND HAVLAK, P. 1997. Programming irregular applications: Runtime support, compilation and tools. *Advances in Computers* 45, 105–153.
- THOZIYOOR, S., MURALIMANOVAR, N., AHN, J. H., AND JOUPPI, N. P. 2008. CACTI 5.1. HP Laboratories.
- VENKATA, S. K., AHN, I., JEON, D., GUPTA, A., LOUIE, C., GARCIA, S., BELONGIE, S., AND TAYLOR, M. B. 2009. SD-VBS: The san diego vision benchmark suite. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC'09)*. IEEE Computer Society, Washington, DC, 55–64.
- WANG, D. ET AL. 2005. DRAMsim: A memory system simulator. *SIGARCH Comput. Archit. News* 33, 4, 100–107.

Received April 2013; revised November 2013; accepted November 2013