Branch-Aware Loop Mapping on CGRAs

Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, Tempe, AZ
{mahdi, aviral.shrivastava, vrudhula}@asu.edu

ABSTRACT

One of the challenges that all accelerators face, is to execute loops that have if-then-else constructs. There are three ways to accelerate loops with an if-then-else construct on a Coarse-grained reconfigurable architecture (CGRA): full predication, partial predication, and dual-issue scheme. In comparison with the other schemes, dual-issue scheme may achieve the best performance, but it requires compiler support – which does not exist. In this paper, we develop compiler techniques to map loops with conditionals on CGRA for the dual-issue scheme. Our experiments show: i) 40% of loops that can be accelerated on CGRA have conditionals, ii) The proposed dual-issue scheme enables our compiler to accelerate loops 40% faster than full predication scheme proposed in [12], and iii) Our compiler assisted dual issue scheme can exploit richer interconnects, if present.

Categories and Subject Descriptors

C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: [Real-time and embedded systems]; D.3.4 [**Processors**]: [Code generation, Compilers, Optimization]

General Terms

Algorithms, Design, Performance

Keywords

Coarse-Grained Reconfigurable Architectures, Compilation, Modulo Scheduling

1. INTRODUCTION

Accelerators have become an indispensable technology for improving the performance and power-efficiency of computation beyond what can be achieved by general-purpose processors. Acceleration is a scheme in which some specific kind of computation can be done faster and in lower power with the use of special hardware (and maybe software). Although special purpose or function specific hardware accelerators (e.g. hardware implementation of FFT) are the best in terms of power-efficiency and performance, they are not programmable, and therefore have limited use. FPGAs (Field Programmable Gate Arrays) allow the flexibility to change the implementation, but their power-consumption is quite high. GP-GPUs (General-purpose graphics processing units) have become very popular recently, since they are relatively easy to program, and can greatly improve performance and power-efficiency of "parallel loops." They can accelerate only parallel loops because their primary method of acceleration is by executing all the iterations of the loop simultaneously. Executing the iterations of non-parallel loops simultaneously can lead to wrong results or significant performance loss.

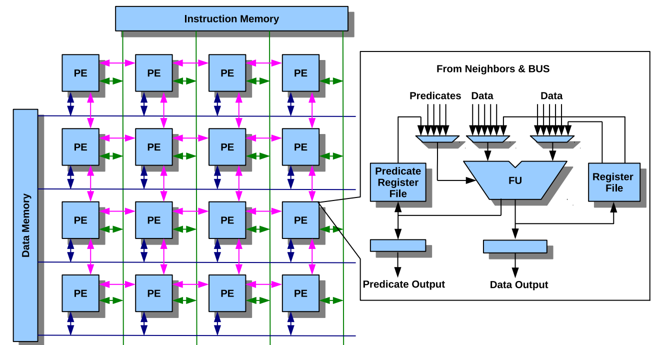

Figure 1: A 4 × 4 CGRA. PEs are connected in a 2-D mesh. A PE is essentially an ALU plus a local register file. It can receive inputs from neighboring PEs output and data bus. An instruction from instruction bus is issued to each PE every cycle, and then PE operates on the data.

Coarse-Grain Reconfigurable Architecture or CGRA is a promising acceleration technology that is not marred by this limitation. A CGRA is simply a network of PEs, with each PE equipped with an ALU and a small register file (see Figure 1). The PEs are connected to neighboring PEs, and the output of a PE is accessible to its neighbors in the next cycle. In every cycle, instructions are issued to each PE, and a row of PEs can access data memory through a common data bus. In contrast to FPGAs, which are programmable at the bit level, CGRAs are programmable at a higher granularity – at the level of arithmetic operations. A CGRA accelerates loops using both "pipelining" (as shown in Figure 2) and "executing loop iterations simultaneously" – therefore it can accelerate both parallel and non-parallel loops.

One of the major challenges associated with all methods of acceleration is the acceleration of loops that have *if-then-else* (ITE) constructs. The fundamental problem is that since the result of the branch is not known before runtime, accelerators use predication to execute the conditional constructs. Hardware accelerators

Permission to make digital or hard copies of all or part 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 bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC'14, June 01 - 05 2014, San Francisco, CA, USA
Copyright 2014 ACM ACM 978-1-4503-2730-5/14/06 $15.00.

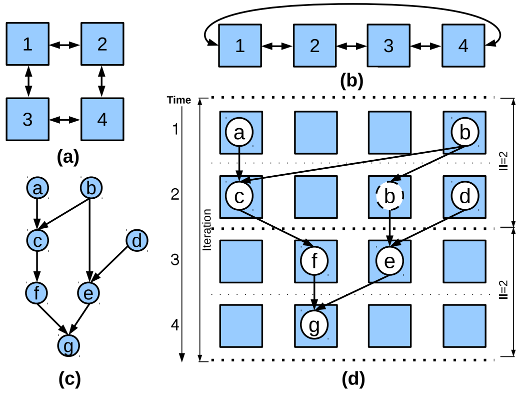

Figure 2: The mapping of a loop kernel on a CGRA. (a) shows an abstract diagram of 2×2 CGRA. (b) shows the linearized form the CGRA. (c) is the data flow graph (DFG) of the loop that we wish to map on the CGRA. (d) shows the mapping of this loop on the CGRA. Note that the mapping is done by pipelining the operations. Also note that the schedule length of the mapping is 4, but the II (Initiation Interval) is 2, since the operations of the next iteration can be started 2 cycles after the start of the current iteration.Therefore the metric of performance is II (lower is better), rather than schedule length.

and FPGAs will execute both the paths of an ITE in parallel, and then choose the results of the taken path. This results in wasted resources and power. GP-GPUs also schedule the instructions in both paths of the ITE, but at the runtime, do not issue the instructions for the not-taken path. This saves power, but the cycles and resources are still wasted. In the graphics processing community, this is referred to as the problem of "branch divergence."

This paper deals with the problem of efficiently executing ITEs on a CGRA. Fundamentally, there are three ways to accelerate loops with an ITE construct on a CGRA. First is full predication - in which operations producing the same output are mapped to the same PE, but at different times. Second is partial predication - in which there is one extra select operation which can be used to merge the values produced in different branches. Third is a dual-issue architecture in which two instructions (one from each side of the ITE) are issued to the PE, and the operation to be executed is chosen at runtime by the PE.

Even though the dual-issue CGRA architecture has the potential to achieve the best performance, full predication and partial predication schemes are more common, since executing loops on dual-issue architecture requires compiler support – and none exists. Specifically, a compiler is needed to merge operations from either branches to be executed on a PE. How operation pairing is performed not only affects the correctness, but also has a significant impact on the resulting performance. In this paper, we formulate and solve the problem of merging operations from the branches to maximize performance. Our results on accelerating loops from various applications from SPEC2006 and digital signal processing benchmark suite demonstrate four significant outcomes. First, it is important to support the acceleration of loops with conditionals. We find that among our benchmarks, more than 40% of the loops that could be accelerated on a CGRA have some form of conditional inside the loop body. Second, our compiler assisted dual-issue scheme can accelerate loops 20% faster than partial predication, and 29% faster than full predication, and 40% faster than the state-of-the-art [12] method. Third, if we add diagonal interconnects in the CGRA, then the acceleration potential of our compiler-assisted dual-issue scheme improves by about 7%. Fourth, after carrying out synthesis and layout of the dual-issue CGRA, we find that the dual-issue architecture only adds 4.7% area and power

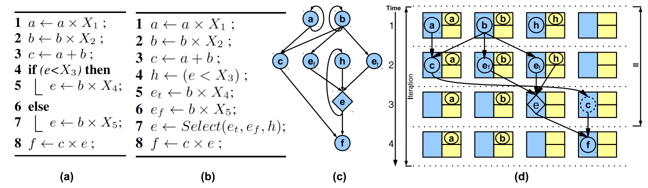

Figure 3: (a) shows a loop body with an ITE. Using partial predication, the loop is transformed to (b). The output e, that is calculated in both the paths is combined using the select operation. If there were outputs that were updated in only one path, we would still need a select operation to choose between the old value and the new one; the new one should be selected only if the path in which the new one is computed is taken. (c) Shows the DFG of the transformed code, and (d) shows the mapping of the DFG on a 2×2 CGRA. Note that the II achieved is 3 cycles.

overhead to the CGRA and about 5.5% frequency penalty over the CGRA with a predication network. This is more than overcome by the significant performs gains we get by better mapping.

2. BACKGROUND AND RELATED WORK

Supporting acceleration of loops with conditionals is important, since many performance-critical loops have ITE constructs in them; ignoring them can greatly reduce the effectiveness of the acceleration architecture. The basic way to support conditionals in accelerators is through predication. Supporting predication on CGRAs requires a predicate network. As shown in Figure 1, the predicate network consists of predicate inputs from the neighbors, a predicate output, and a small predicate register file. The result of the conditional executed on a PE is propagated to the PEs on which operations of the if-part and the operations of the else-part are executed through the predicate network. Most CGRAs implement the hardware of the predicate network [4, 6, 13]. There are three basic ways to support acceleration of conditionals on CGRAs.

2.1 Partial Predication

In partial predication, the operations of both the if-part and the else-part are mapped on different PEs. If the same variable is being updated in both the if-part and the else-part, the final result is computed by selecting the output from the path that must have been executed based on the evaluation of the branch condition. This is achieved through a special operation, called *select* or a *conditional move*, which takes in the result of the branch condition (through predicate network), and the two outputs to select the correct one. If a variable is updated in only one path, a select operation is still required to choose between the old value (before the ITE, or even the value from previous iteration(s)) and the new value. The new value is valid only if the path of the branch in which the new value is computed should have been executed. Architectural support and the idea of partial predication was presented in [11].

Figure 3(b) shows the partial predication transformation of the loop body in figure 3(a). In this scheme, new variables for operations in ITE paths are introduced. Therefore, operations at line 5 and 6 in Figure 3(b) can be executed independently. At the end, predicate (h) chooses the final value of (e). The select instruction is necessary to support partial predication transformation. Figure 3(c) is the DFG generated from partial predication transformation and 3(d) shows the mapping of this loop to a 2×2 CGRA. II in this mapping is 3 and is the minimum possible for this transformation.

To map an ITE that has "n" operations on each path, the number of operation nodes for partial predication is, in the worst case, $3n$. This is because all the operations from both paths must be mapped $(2n)$, as well as the select operations (n). A select operation is needed for each output that will be needed in the rest of the

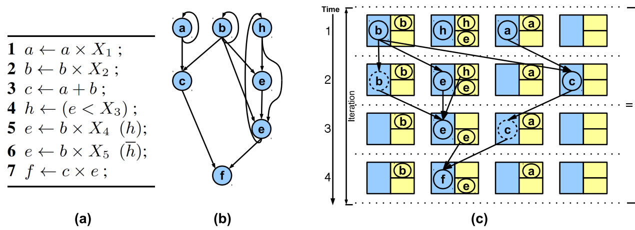

Figure 4: (a) shows the transformed code using full predication scheme. There is no select operation, but operations at line 5 and 6 that compute the variable "e" have to be mapped to the same PE. If there was some variable that was updated in only one path, then the operation computing that variable must be mapped to the PE that previously computed the variable. Note that in addition to the DFG, there are placement constraints that must be met for a legitimate mapping for full predication. (b) shows the DFG of the transformed loop. and (c) shows the mapping of the loop on a 2×2 CGRA.

program. The select operation may not be required for intermediate outputs. In the worst case, all the $2n$ outputs will be used in the rest of the program, and therefore n select operations will be required. This increases the II and results in a loss of performance.

2.2 Full Predication

Full predication does not require a select operation. Instead, the operations that update the same variable are mapped to the same PE, albeit at different times. Since only one of the operations will be executed at runtime (and the other will be squashed), the correct value of the output is present in the PE after the maximum time. If an output is computed in only one path of the ITE, then the output must be computed on the PE that previously updated the same variable. This is done so that after ITE, for each variable there is a unique PE, that has its value and therefore no select operation is needed. The idea of full predication was presented in [12].

Consider the body of a loop shown in figure 3(a). The result of full predication transformation is presented in figure 4(a). Operations at line 5 and 7 in the original snippet of the code are guarded by (h) in figure 4(a) at line 5 and 6. Operation at line 4, uses variable (e) that is updated in the previous iteration. To make sure that variable (e) gets updated properly, operations at line 5 and 6 must be mapped to the same PE where variable (e) is kept. Figure 4(c) presents the best mapping of this loop after full predication transformation. The best II achieved for full predication is 4. If we have to map only an ITE that has n operations on either path, then the number of operation nodes for full predication DFG in the worst case is $2n$, but there are placement constraints for each of $2n$ nodes. The tight constraints on the operation placement are very restrictive, and severely degrade the performance.

2.3 Dual-Issue

This scheme alleviates the problem of accelerating conditionals by issuing two instructions to a PE simultaneously, one from the if-path, and one from the else-path. Depending on the result of the conditional operation, only one of them is executed at runtime. This method also does not require select operation. The idea of dual-issue scheme was presented recently in [11].

Figure 5(a) shows the DFG after partial predication transformation. The nodes on either side of the DFG are merged to make a packed node. Packed node represents dual-issue operations. Operation (e) represented by a hexagonal shape node is a packed node. The adjacency of nodes are preserved after this transformation. Figure 5(b) shows the DFG after the packing transformation. Figure 5(c) shows the mapping after this transformation to a 2×2 CGRA. II of this mapping is 2 and is the minimum possible. If we have to map only an ITE that has n operations on either path, then

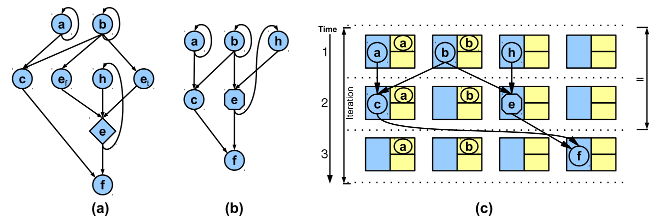

Figure 5: (a) shows the DFG with partial predication transformation. The operations from the two sides of the branch are merged to form packed nodes (packed nodes represent a pair of nodes executed in a dual-issue PE). (b) shows the DFG after the packing transformation. (c) shows the mapping of the transformed DFG on a 2×2 CGRA. II of 2 is achieved.

the number of operation nodes for dual-issue DFG in the worst case is n. Plus there are no placement constraints. Therefore, we believe dual-issue may be the best solution to accelerate conditional loops.

3. CONTEXT AND CONTRIBUTION OF THIS PAPER

Although the dual-issue scheme promises the best performance, partial and full predication seem to be more common in CGRAs. The reason is that the traditional schemes of full and partial predication do not require much change in the compiler. However, the new scheme of dual-issue requires extensive compiler support. This is because supporting partial predication requires generation of select operations, which is a well studied compiler topic. In fact it is a part of Single Static Assignment (SSA) transform that is present in most compilers, and the select or conditional move operations are constructed in the *phi elimination* pass. Once the DFG with select operations is generated, existing CGRA scheduling and mapping techniques (e.g., [1, 2, 5, 7, 8, 9, 10, 14, 15, 18, 20, 21]) designed to map non-conditional loops on CGRA, can also be used. Supporting full predication is also relatively straightforward, since the DFG remains the same. The only new aspect is placement constraints on the operations, which only means that as soon as an output of an operations is mapped, the mapping of the second operation updating that output is also fixed.

On the other hand, for a dual-issue architecture, compiler support is needed to pack operations from both paths of a branch into a packed node. How we do operation pairing not only affects the correctness, but also has a significant impact on the resulting performance. In this paper, we formulate the conditions for legitimate packing, and also formulate the problem of performance optimization by packing, and then solve the problem to achieve efficient mapping for a dual-issue architecture.

Supporting execution of conditional loops in CGRAs has not received much attention in the research community. The only compiler technique we have seen is a form of full predication, presented in [12]. In this scheme, operations within body of an if-then-statement are mapped to the same PE. We believe that this is too restrictive, and it will cause significant performance loss because other PEs will not be utilized. It is important to note that the performance is proportional to PE utilization.

4. PROPOSED APPROACH

In this section, we present our mapping framework, BRMap. BRMap starts with a given control flow graph (CFG) of a loop. BRMap enables user to choose between full predication, partial predication, and dual issue transformations to be conducted. Based on selected transformation, BRMap constructs a DFG from the input CFG. At the end, BRMap calls REGIMap [10] to complete the mapping.

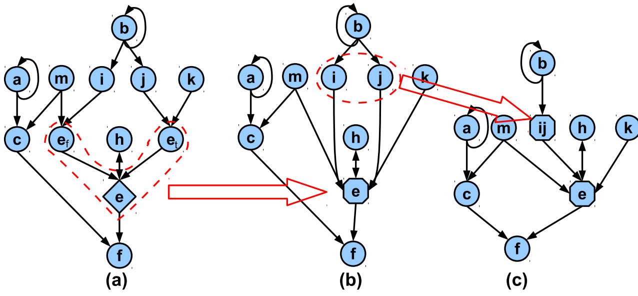

Figure 6: (a) shows the input DFG. The first candidate to form a packed node is select instruction e and its inputs. (b) select node and its inputs are replaced with a new packed node. Nodes i and j are the best pair to form next packed node. Nodes k and m cannot be reduced because m is used as an input to instruction c that is not within the loop body. (c) shows the minimized DFG. The number of nodes in the final DFG is reduced by three.

Step 1: DFG Construction. BRMap constructs a DFG from the CFG of a loop first. There are standard schemes to construct hyperblocks from multiple basic blocks of a CFG using full-predication and partial-predication transformations [19]. DFG constructed from full-predication transformation can be directly fed to the underlying CGRA mapping algorithm. However, it is necessary to ensure that all instructions updating same variable are to be mapped on the same physical PE.

DFG constructed after partial-predication transformation requires minimal change in the mapping algorithm. REGIMap only allocates registers for nodes on PEs. We enhance REGIMap to allocate registers at PEs and predication network using the same technique.

Algorithm 1: Minimize(Input D)

1. **begin**
2.   **while** $|M|$ *increasing* **do**
3.   **while** $|C|$ *increasing* **do**
4.   **for** *All instructions o in D* **do**
5. $S_o \leftarrow$ successors of o;
6.   **if** $S_o = S_o \cap M$ *or o is a select* **then**
7. $C \leftarrow C \cup \{o\}$;
8.   ASAP_Schedule D;
9.   ALAP_Schedule D;
10.   **for** *All instructions o in C* **do**
11.   **if** *o is a select instruction* **then**
12. $I_i^o \leftarrow$ if-path input of o ;
13. $I_e^o \leftarrow$ else-path input of o ;
14.   **if** *o is only successor of* I_i^o *and* I_e^o **then**
15.   Pack o, I_i^o, and I_e^o into P^o ;
16. $M \leftarrow M \cup \{P^o\}$;
17.   **else**
18. $S \leftarrow$ (if-path$\times$ else-path) inputs of o;
19.   Sort S by cost of each pair;
20.   **if** *the best cost is positive* **then**
21.   Replace selected pair with P^o ;
22. $M \leftarrow M \cup \{P^o\}$;

Dual-issue scheme starts from a partial predicated DFG. DFG is scheduled first and $MII = Max(ResMII, RecMII)$ is extracted using EPIMap algorithm [9]. Before conducting any DFG minimization, we determine whether minimizing DFG by forming packed instructions would benefit the performance of a mapping or not. BRMap conducts DFG minimization only if there is a performance benefit to do so. In some DFGs, reducing the number of nodes may not benefit the performance at all. The reason is that, mapping II in some loops is limited by $RecMII$ rather than $ResMII$. Although packing pairs of instructions decreases the total number of nodes in a DFG, it does not affect $RecMII$ at all. It is because reducing the number of nodes through packing does not alter the latency of any path. Therefore, such loops would not benefit from reducing the number of operations because their performance is limited by latency of critical cycles [22]. The second reason is that even if the number of nodes in a DFG is reduced, the number of reduced nodes can be insufficient to decrease II. Given a DFG $I = (V_I, E_I)$ and an $M \times N$ CGRA, $ResMII = \lceil \frac{|V_I|}{M \times N} \rceil$. If removing few nodes from DFG does not alter $ResMII$ (because of the non-linear relation between $|V_I|$ and MII), packing is unlikely to benefit performance. In this case, BRMap conducts a preliminary mapping first. If the underlying CGRA mapping algorithm finds a mapping with an $II > MII$, only then BRMap starts packing instructions to reduce the number of operations in DFG. When CGRA mapping algorithms fail to find a mapping for a given II, extra nodes (in form of routing and/or recomputing nodes) are added to the DFG [9]. Those extra nodes relax data dependencies between instructions whose data dependencies could not be satisfied in a mapping. However, increasing the number of nodes in a DFG gradually increases $ResMII$ and consequently MII (note that it is non-linear relation). For such cases, reduction in the number of nodes through packing instructions provides further flexibility to the mapping algorithm to add routing and recomputing nodes when a mapping failure occurs.

Step 2: Packing Pair of Nodes. The minimization algorithm is presented in Algorithm 1. To conduct DFG minimization, BRMap first schedules the operations. Scheduling determines a partial order for nodes to execute. This order is essential to form dual-issue instructions. If nodes are packed without respecting the partial order of operations in a DFG, it is possible to transform the input DFG to the one for which no feasible schedule exists. Thus, it is crucial to respect partial orders of instruction and pack them carefully. Consider the following paths in a DFG. $P_1 = (i_1, i_2, s)$ and $P_2 = (i_3, i_4, s)$ and $P_3 = (i_5, s)$. i_5 is the predicate Boolean input and s is a select instruction. If pairs $C_1 = (i_1, i_4)$ and $C_2 = (i_2, i_3)$ are chosen, there is no feasible schedule for the transformed DFG. It is because i_1 must be scheduled after i_2 which implies $C_1 < C_2$. However, i_3 must be scheduled before i_4 which implies $C_2 < C_1$. Thus, no feasible order for C_1 and C_2 exists.

To respect partial order of operations, BRMap starts from a select instruction. Each select instruction has three inputs: an input from if-path of an ITE construct, another input from else-path, and a Boolean input to choose among former two. A select instruction along with two inputs from if-path and else-path are the first candidates to form a packed node. The necessary condition to form a packed node is that the schedule window ($t_{ALAP} - t_{ASAP}$, i.e. time between ASAP schedule till ALAP schedule) of the pair overlaps. Please note that if only one of those operations are to be executed at run-time, there is no need to have a select operation.

In Figure 6(a), select instruction e and its inputs are packed to form a new node. The transformed DFG is shown in Figure 6(b) where three nodes are merged.

Next, BRMap finds the set of input nodes to the packed nodes. Similar to select instruction, there are inputs from if-path and else-path of an ITE constructs to those packed nodes. However, the number of inputs of packed nodes may vary. This is different from select instruction where the number of inputs is always three. At

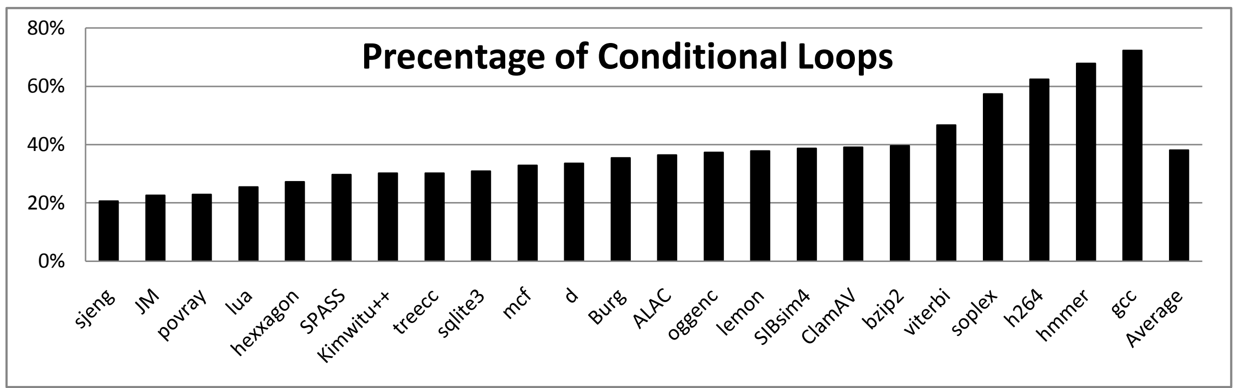

Figure 7: It is important to support loops with conditional constructs. Many important loops have at least one conditional clause in their body.

this step, we need to ensure that for any pair of instructions we choose, there is an instruction from if-path and one from else-path. For instance, in Figure 6(b), j and k cannot form a pack because they both are within the if-path. If there are many possible pairs, BRMap attempts to pack a pair that it is easier to place at the mapping step. Let I_a be the set of inputs of node a. Consider two candidate nodes to form a packed node (a, b). BRMap finds the intersection of I_a and I_b. It also finds the overlap in schedule window of a and b. BRMap finds a pair with the maximum intersection and overlap in schedule windows. BRMap packs any pair of nodes that is possible to pack in a greedy manner ($O(n^2)$). In Figure 6(b), nodes i and j in this DFG are packed next. They share input b and they are scheduled at the same cycle. The minimized DFG is shown in Figure 6(c). Nodes m and k cannot be packed because node c is not within an ITE construct and there is an arc from m to c. A packed node from a pair of nodes with common inputs is easier to map. For a packed node, the number of dependencies that must be satisfied to find a valid mapping is usually higher than a regular node. When a pair of nodes forming a packed node share an input, the number of dependent nodes that should be placed in neighbering PEs decreases. Therefore, it is easier to place such packs compare with the one without any common input. BRMap iteratively finds pairs of nodes to form packed nodes until no further pair can be found ($O(n^3)$).

Step 3: Mapping Transformed DFG onto CGRA. The number of data dependencies after forming packed nodes makes mapping of DFG significantly more challenging than a regular DFG. Thus, it is important to use a constructive CGRA mapping technique. REGIMap is a constructive CGRA mapping algorithm which efficiently utilized resources on CGRA. After DFG minimization, BRMap calls REGIMap [10] to find a valid mapping of DFG on CGRA. We modified REGIMap to allocate registers both in PEs register files and predication network.

5. EXPERIMENTAL RESULTS

We have integrated our compiler technique as a separate pass in the llvm compiler framework [17]. We have modeled CGRA as an accelerator in Gem5 system simulation framework [3]. Loops that are important for performance are selected from SPEC2006 [16] and digital signal processing applications. We conduct our experiments to explore the performance of the various architectural and compiler techniques for handling conditionals in CGRA. We map the loops on a 4×4 CGRA with sufficient instruction memory to hold all instructions within a loop body as well as sufficient data memory space to hold all the variables. Latency of all operations are assumed only one cycle. Load and store operations requires two CGRA operations, one for address bus transaction and the other for data bus transaction. The bus is shared among all PEs within a row; in other words, only one memory transaction can proceed at any cycle in a row. We conduct our experiments on mesh-interconnected CGRA and then we enrich interconnection further with diagonal connections between PEs.

5.1 Need for supporting Conditionals in Loops

Our first evaluation demonstrates the importance of supporting conditions within loops. If conditional constructs are not supported, many performance-significant loops cannot be accelerated on CGRAs. As shown in Figure 7, about 40% of loops that can be executed on CGRA have at least one ITE construct within the body of the loop. Here, the condition we are referring to is different from the main loop condition which controls the number of times a loop would be executed. We are referring to an ITE construct in the loop. This plot is constructed at -O3 optimization in llvm.

5.2 Dual-issue architecture with our compiler technique outperforms all other forms to handle conditionals

Next we compare different techniques to accelerate loops with conditionals, namely: full predication approach presented in [12], full predication, partial predication, and our approach for dual-issue. Figure 8 plots the achieved II of the loops by different schemes. The bars on the right corner show the average II achieved over all the loops by the techniques. Figure shows that the dual-issue approach leads to the best acceleration (least II) among all the techniques. The full predication approach presented in [12] performs the worst, since it is highly restrictive – all the instructions inside the conditionals have to be mapped to the same PE – this results in long schedule length, and long II. Our approach for full predication performs better, primarily because it is less restrictive. The restriction is that the operations in different branches that are generating the same output must be mapped to the same PE. Partial predication performs better than both of these, since it does not add restrictions in mapping, only adds more nodes to the graph. However, dual-issue scheme performs best, since it neither adds restrictions, nor extra nodes in the graph. Overall dual-issue architecture can improve II by almost 42% as compared to the full predication scheme proposed in [12].

5.3 Performance advantage of dual-issue approach scales with interconnect network

As interconnection is enriched with diagonal connections, the full predication scheme presented in [12] does not improve considerably (only about 0.7% on average). This is because this technique constrains ITE constructs to be executed sequentially, and does not benefit from either more PEs nor richer interconnect between PEs. Full predication gains the most performance benefit from diagonal connections, (about 7% on average). This is because many instructions in this scheme requires three dependencies to be satisfied in mapping. Because of the high data dependency between operations, it is more likely that all dependencies cannot be satisfied in mapping in a mesh-interconnected CGRA. Therefore, mapping fails and more routing nodes are needed to be inserted in the DFG.

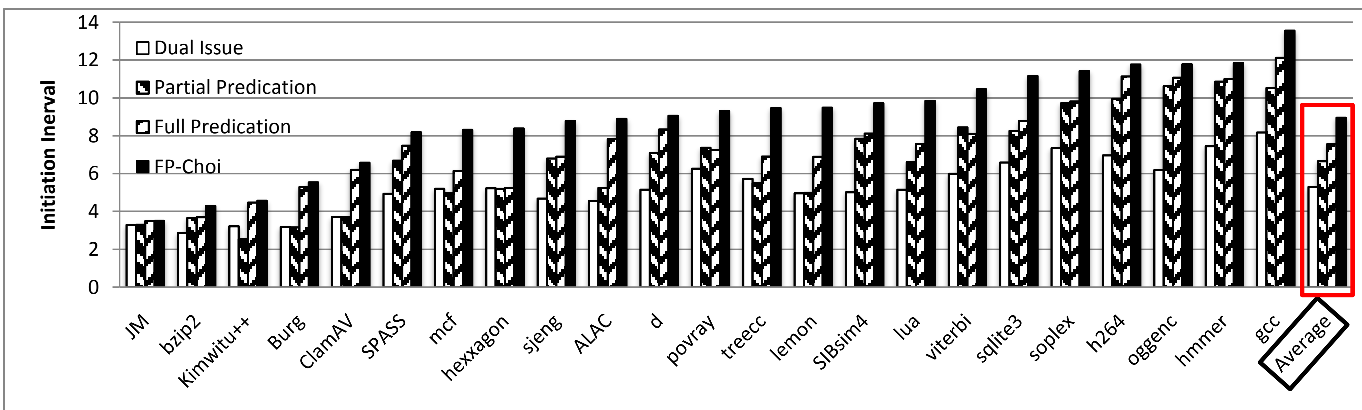

Figure 8: This figure plots the achieved II for different loops with conditionals from various benchmarks. The graph shows that Dual-issue architecture with our proposed compiler technique results in the lowest II. Performance is inversely proportional to II.

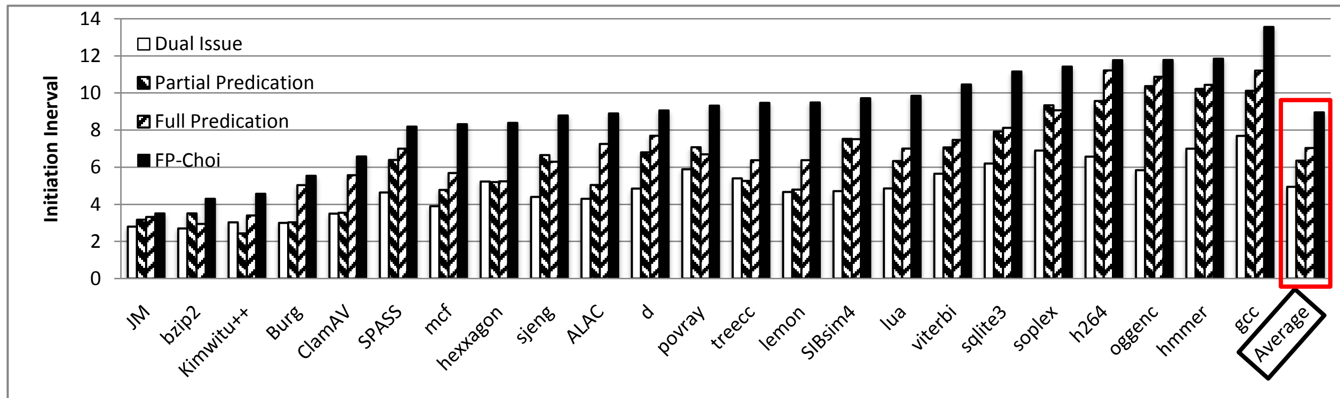

Figure 9: The Performance of compiled loops using different acceleration techniques in mesh and diagonal interconnected CGRA. Dual-issue scheme leads to the best acceleration among all. Partial predication and full predication show relatively close performance benefit. Better interconnection benefits all application but the benefit is narrow for loops limited by $RecMII$.

This eventually leads to more frequent increment in II. Partial predication achieves modest benefits from better interconnection (on average about 4.5%). Dual-issue architecture gains 6.7% performance benefit from better interconnection because dual-issue instructions have the highest data dependency among all instructions. Therefore, when more communication channels are available, data dependencies are more likely to be met in mapping. However, even in a richly connected CGRA, dual-issue architectures achieves the lowest II, and (therefore) the best performance.

6. SUMMARY

Coarse-Grain Reconfigurable Architecture or CGRA is a promising acceleration technology to accelerate loops. However, it is challenging to accelerate loops that have if-then-else constructs. In this paper, we study different acceleration schemes for loops with if-then-else constructs and develop compiler techniques to efficiently accelerate such loops on CGRAs.

7. REFERENCES

[1] Ahn, M., Yoon, J., Paek, Y., Kim, Y., Kiemb, M., and Choi, K. A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In *Proc. DATE* (2006), pp. 363–368.

[2] Bansal, N., Gupta, S., Dutt, N., Nicolau, A., and Gupta, R. Network topology exploration of mesh-based coarse-grain reconfigurable architectures. In *Proc. DATE* (2004), pp. 474–479.

[3] Binkert, N., Beckmann, B., et al. The gem5 simulator. *SIGARCH Comput. Archit. News 39*, 2 (Aug. 2011), 1–7.

[4] Chang, K., and Choi, K. Mapping control intensive kernels onto coarse-grained reconfigurable array architecture. In *Proc. ISOCC* (2008), pp. I–362–I–365.

[5] Chen, L., and Mitra, T. Graph minor approach for application mapping on cgras. In *Proc. FPT* (2012), pp. 285–292.

[6] De Sutter, B., Raghavan, P., and Lambrechts, A. *Handbook of Signal Processing Systems*, 2 ed. Springer, 2013, ch. Coarse-Grained Reconfigurable Array Architectures, pp. 553–592.

[7] Dimitroulakos, G., Galanis, M., and Goutis, C. Exploring the design space of an optimized compiler approach for mesh-like coarse-grained reconfigurable architectures. In *Proc. IPDPS* (2006), pp. 113–122.

[8] Friedman, S., Carroll, A., Van Essen, B., Ylvisaker, B., Ebeling, C., and Hauck, S. Spr: an architecture-adaptive cgra mapping tool. In *Proc. FPGA* (2009), pp. 191–200.

[9] Hamzeh, M., Shrivastava, A., and Vrudhula, S. Epimap: using epimorphism to map applications on cgras. In *Proc. DAC* (2012), pp. 1284–1291.

[10] Hamzeh, M., Shrivastava, A., and Vrudhula, S. Regimap: register-aware application mapping on coarse-grained reconfigurable architectures (cgras). In *Proc. DAC* (2013), pp. 18:1–18:10.

[11] Han, K., Ahn, J., and Choi, K. Power-efficient predication techniques for acceleration of control flow execution on cgra. *ACM Trans. Archit. Code Optim. 10*, 2 (May 2013), 8:1–8:25.

[12] Han, K., Choi, K., and Lee, J. Compiling control-intensive loops for cgras with state-based full predication. In *PRoc. DATE* (2013), pp. 1579–1582.

[13] Han, K., Paek, J. K., and Choi, K. Acceleration of control flow on cgra using advanced predicated execution. In *Proc. FPT* (2010), pp. 429–432.

[14] Hannig, F., Dutta, H., and Teich, J. Regular mapping for coarse-grained reconfigurable architectures. In *Proc. ICASSP* (2004), pp. 57–60.

[15] Hatanaka, A., and Bagherzadeh, N. A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In *Proc. IPDPS* (2007), pp. 1–8.

[16] Henning, J. L. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News 34*, 4 (Sept. 2006), 1–17.

[17] Lattner, C., and Adve, V. LLVM: A compilation framework for lifelong program analysis and transformation. pp. 75–88.

[18] Lee, G., Lee, S., and Choi, K. Automatic mapping of application to coarse-grained reconfigurable architecture based on high-level synthesis techniques. In *Proc. ISOCC* (2008), pp. 395–398.

[19] Mahlke, S. *Exploiting instruction level parallelism in the presence of conditional branches*. PhD thesis, UIUC, 1997.

[20] Mei, B., Vernalde, S., Verkest, D., De Man, H., and Lauwereins, R. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proc. DATE* (2003), pp. 296 – 301.

[21] Park, H., Fan, K., Mahlke, S. A., Oh, T., Kim, H., and Kim, H.-s. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proc. PACT* (2008), pp. 166–176.

[22] Rau, B. R. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proc. MICRO* (1994), pp. 63–74.