Compiling Control-Intensive Loops for CGRAs with State-Based Full Predication

Kyuseung Han and Kiyoung Choi
School of EECS
Seoul National University, Seoul, Korea
{ darprin, kchoi }@dal.snu.ac.kr

Jongeun Lee
School of ECE
UNIST, Ulsan, Korea
jlee@unist.ac.kr

***Abstract*—Predication is an essential technique to accelerate kernels with control flow on CGRAs. While state-based full predication (SFP) can remove wasteful power consumption on issuing/decoding instructions from conventional full predication, generating code for SFP is challenging for general CGRAs, especially when there are multiple conditionals to be handled due to exploiting data level parallelism. In this paper, we present a novel compiler framework addressing central issues such as how to express the parallelism between multiple conditionals, and how to allocate resources to them to maximize the parallelism. In particular, by separating the handling of control flow and data flow, our framework can be integrated with conventional mapping algorithms for mapping data flow. Experimental results demonstrate that our framework can find and exploit parallelism between multiple conditionals, thereby leading to 2.21 times higher performance on average than a naive approach.**

***Index Terms*—CGRA; reconfigurable architecture; predication; predicated execution, conditional, compilation;**

I. Introduction

Coarse-grained reconfigurable architectures (CGRAs) [1]–[6] are a promising solution for embedded systems, since they can provide high performance, low power, and flexibility at the same time. High performance can be provided through abundant processing elements (PEs). Compared to multi-core architectures, CGRAs can consume much lower power while maintaining the flexibility of reconfiguration since each processing element is much simpler and a single controller manages all PEs.

However, the use of a single controller to achieve high energy efficiency creates challenges in terms of accelerating programs with control flow. Due to the lack of hardware in CGRAs that can directly handle branch operations, even a small fragment of code with control flow may require intervention by a main processor, or it cannot be executed on a CGRA at all. To remedy this problem, many CGRAs adopt predicated execution [7]–[9], which essentially converts control dependence into data dependence.

To reduce energy wasted by conventional predication due to fetching/decoding instructions, a low power predication technique named state-based full predication (SFP) has been recently proposed [9]. It avoids the capacity and power overhead in instruction memory incurred by the additional field in the predicated instructions of the conventional approach. It also saves power by turning off PEs on untaken paths, which is not allowed in the conventional one. As a result, SFP can save energy by up to 23.9%.

However, there are a number of challenges in compiling loops for CGRAs that support SFP. At the core of the problem lies operation-to-PE binding, which would be trivial if the target architecture has only one PE, or allows SIMD execution only as in [9]. But if the target architecture allows non-SIMD execution, that is, different operations can be performed by different PEs in the same cycle as in a VLIW processor, the ways to map operations to multiple PEs can affect the performance of the CGRA significantly. If operations in one conditional are scattered among multiple PEs, the overhead of managing the state registers of PEs may exceed the benefit. Also, if operations from several conditionals are interleaved in their schedule on the same PE, switching the state register will cause large overhead.

Another issue arises from the fact that power saving mode of a PE renders almost all the resources of a PE including the local register file of a PE inaccessible. If a PE in a power-saving (sleep) state has a scalar variable stored in its register file, and the variable is needed by another PE, we must route the variable in advance before the first PE goes into the power-saving state. Otherwise, routing must be processed after exiting the power-saving mode so performance will be greatly degraded. Even worse, if another routing is required in the opposite direction (i.e., the PE in sleep state wakes up when the data from the other PE is available) at the same time, then the execution can go into deadlock and will be failed.

This paper presents a mapping framework for SFP-based CGRA processors that allow non-SIMD execution in order to maximize performance while guaranteeing correctness.

II. Related Work

Many CGRAs have been proposed so far [1]–[6]. Although each of them has its own merits, adding SIMD features to the architecture has been commonly claimed to be one of the most important merits [1]–[3], [6]. It is because SIMD is already well known to be effective and thus it is widely used in commercial processors.

Supporting control flow on data-level parallelism processors including SIMD processors as well as CGRAs has been addressed continuously [7]–[11]. To the best of our knowledge, predication is the only solution known so far; some researches have focused on automating the use of conventional predications and some others have invented a new form of predications. For automatic mapping of control flows on CGRA using conventional predications, [8] and [11] have shown that only minimal efforts are needed. By adopting *if*-conversion (e.g., [12]), control dependency can be converted to data dependency. As a result, conventional mapping algorithms do not need to handle control flow explicitly, but only need to consider data dependency as usual. [9] has first proposed an approach to low power predication at the architectural level. However, compilation issues have not been addressed yet.

III. Compilation for State-Based Full Predication

A. Target Architecture

Our target architecture consists of a 2D array of PEs, which supports multiple modes of execution: full SIMD, partial

```
for (i=0; i<8; i++)
{
  if(c[i] == 1)      load R0 c[i] load R0 c[i]
  {                  cmp R0 #1    cmp R0 #1
    x[i] = 0;        b neq pc+4   csleep neq #3
    y[i] = 1;        store 0 x[i] store 0 x[i]
  }                  store 1 y[i] store 1 y[i]
  else               b uc pc+3    csleep uc #2
  {                  store 1 x[i] store 1 x[i]
    x[i] = 1;        store 0 y[i] store 0 y[i]
    y[i] = 0;
  }
}
         (a)                (b)          (c)
```

Fig. 1. (a) an example of loop code, (b) assembly code of the loop body using branchs, and (c) assembly code of the loop body using SFP.

SIMD, and MIMD. In full SIMD, all PEs execute the same instruction (or have the same configuration) at a time. There can be various implementations of partial SIMD. For example, PEs on the same column can have different instructions, but PEs on the same row execute the same instructions. In MIMD, all PEs can execute different instructions at the same cycle. Since using SFP in full SIMD mode is trivial (Section III-B), we assume in this paper that our target architecture is used in partial SIMD or MIMD mode. Partial SIMD and/or MIMD modes are frequently exploited by many CGRAs including MorphoSys [3], REMARC [2], PADDI [1], and FloRA [6], [13].

SFP mechanism is implemented on PEs as suggested in [9]. Each PE is in either of *awake* and *sleep* states. Instructions are executed normally in the awake state but aborted in the sleep state. The transitions between the two states are made by sleep instructions and the sleep period counter.

B. Mapping Issues on CGRAs

If only one PE (or full SIMD mode of our target architecture) is considered for mapping one iteration of a loop, generating code for SFP will be very straightforward. Simple replacement of branch instructions by sleep instructions can generate correct and efficient code for SFP [9]. Fig. 1 shows an example of original C code, the corresponding assembly code obtained through compilation using branch instructions, and the one obtained by using SFP.

For CGRAs that use partial SIMD and/or MIMD[1], however, there are a number of compilation issues as mentioned in Section I, which arise because (1) multiple PEs can be used to map one iteration and (2) a PE can interleave the executions of multiple conditionals. To ensure correctness and also maximize performance, it becomes clear that one should not shuffle operations from different conditionals. In the next section we present our technique to run them in parallel while keeping them separated.

Although multiple conditionals in one loop may seem rare in the initial code, simple loop transformations such as loop unrolling can multiply the number of conditionals. Since such multiple conditionals can be performed at the same time, which is very common, mapping them properly is especially important to maximize the effectiveness of loop unrolling.

IV. Proposed Mapping Framework

A. Overall Flow

Fig. 2 illustrates the overall flow of our framework for mapping loops with control flow on SFP-based CGRAs. It starts from IR (intermediate representation), which can be obtained easily by frontend tools. The framework largely consists of two parts. The first part converts IR to CDFG (control data flow graph), extracting parallelism on the way. The second part takes the CDFG and allocates PEs to different parts of the CDFG (each part is a DFG) so that each part can be mapped separately in a temporal or a spatial manner using known mapping algorithms.

[1]There is no difference between partial SIMD and MIMD modes during mapping in that several PEs is considered for mapping one iteration.

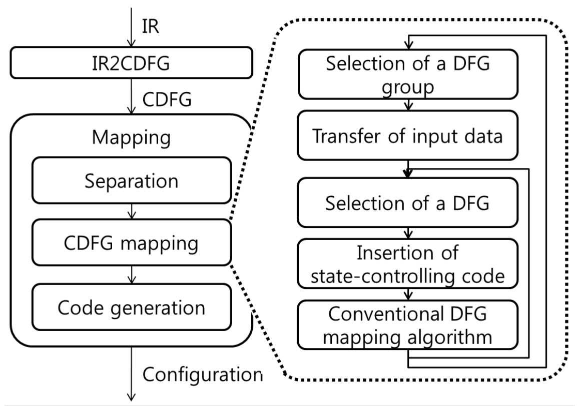


Fig. 2. The mapping framework on SFP-based CGRAs.

B. From IR to CDFG

1) CDFG Generation: The IR for a loop body is given as a CFG of DFGs, where each node (DFG) represents a basic block. Fig. 3a illustrates the CFG of a loop body, which contains one *nested-if* construct followed by a simple *if-else*.

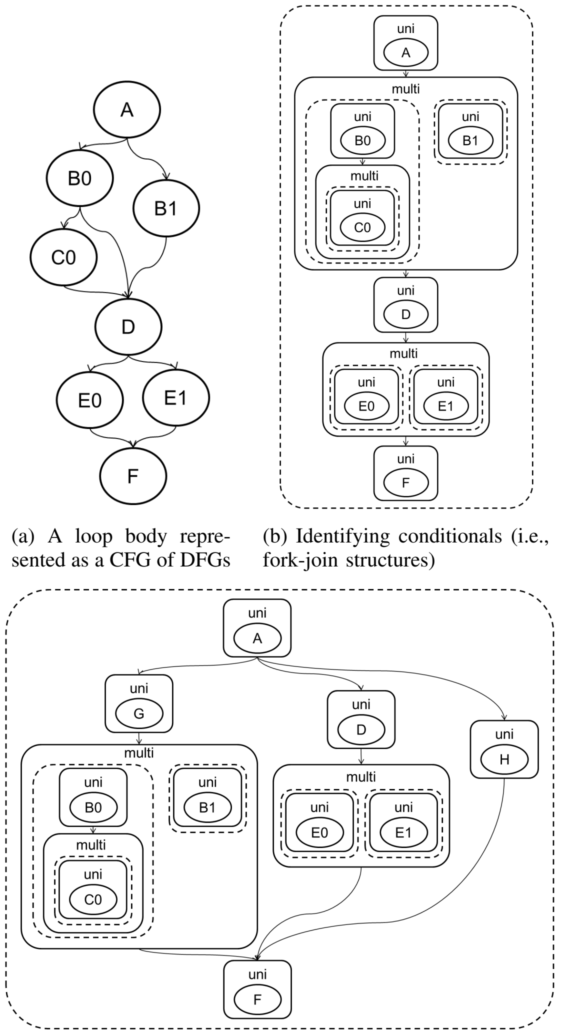
We first transform the CFG to our CDFG representation so that the control structure and parallelism can be captured more explicitly. We use a hierarchical CDFG defined as follows. Each node of the CDFG is a block of either of two types: *unipath* and *multipath*. A unipath block is simply a DFG, whereas a multipath block contains one or more CDFGs with a condition for each CDFG. Fig. 3b illustrates the CDFG corresponding to the IR in Fig. 3a. In the figure, ovals, solid round boxes, and dashed ones represent DFGs, blocks, and CDFGs, respectively. Directed edges indicate data dependency between two blocks. Note that the edges in Fig. 3b are obtained through data flow analysis and are different from those in Fig. 3a.

To transform an IR to a CDFG representation, we first identify conditionals to generate CDFGs at lower levels of hierarchy (see Fig. 3b). Since multipath blocks can contain other multipath blocks in such lower level CDFGs, *nested-if* structures can be naturally represented.

2) Exploiting Parallelism: We then update data dependence among blocks, which may reveal parallelism between blocks. In Fig. 3b, if DFGs D, $E0$, and $E1$ are not dependent on $B0$, $B1$, or $C0$, we can remove the dependence edge between them, making D an immediate successor of A as illustrated in Fig. 3c. In addition, we exploit more parallelism by extracting operations from A or F if they have no dependency with any operation in the conditionals. We separate those operations out as new DFGs (G and H) as shown in Fig. 3c.

C. Separation

To ensure correctness and maximize performance, we map operations from different conditionals separately either in temporal or spatial manner, which can be achieved by DFG grouping and PE-to-DFG allocation. A DFG group is defined as a set of DFGs running in parallel. We group DFGs and order groups as a list. Within a group, we allocate different PEs to



(a) A loop body represented as a CFG of DFGs

(b) Identifying conditionals (i.e., fork-join structures)

(c) Exploiting parallelism

Fig. 3. Conversion process from IR to CDFG.

different DFGs, which corresponds to spatial separation. The groups are put into a list in the order of their generation, which corresponds to temporal separation.

When grouping DFGs, we need to consider two aspects. If many DFGs are put into one group so that they can run in parallel, it can help increase performance especially when each DFG has low instruction-level parallelism. On the other hand, if too many DFG are in one group, registers can be spilled, resulting in performance reduction. Thus our strategy is to assign just enough number of PEs to each DFG to avoid register spills and then to group DFGs as long as PEs are available. Therefore we first estimate the register requirement of each DFG, followed by DFG grouping and PE-to-DFG allocation.

1) Register Requirement Estimation: The register requirement of a CDFG is the maximum of all the register requirements of its blocks. For a unipath block, its register requirement is that of its DFG. For a multipath block, its register requirement is the maximum of those of its CDFGs. Thus we only need to obtain the register requirement of DFGs. To calculate the upper bound of the number of registers needed for a DFG, we run a scheduler on the DFG without doing detailed mapping such as operation-to-PE binding and register binding. This is because scheduling is enough to calculate the upper bound of the register usage (in our target architecture, the binding processes do not affect the register usage).

2) DFG Grouping and PE-to-DFG Allocation: Based on register requirements, we calculate PE requirements of *ready* DFGs considering available number of registers. If an architecture has four PEs and total eight registers are available now and if a DFG requires three registers, then the PE requirement of the DFG is two since there are two available registers per PE on the average (in the later scheduling and binding phase, if the actual number of available registers is different from the average, then we may have to take extra cycles to move data around). After that, grouping is performed in a way similar to heaviest-first selection in the knapsack problem. That is, DFGs are selected with most PE requirement first and packed into a sack while the capacity of the sack is not exceeded. After packing one sack, we update the number of registers that will be available after the execution of the previous group. And then we pack another sack by repeating the above processes.

D. CDFG Mapping

CDFG mapping flow is shown in Fig. 2 surrounded by the dotted line. Differing from conventional DFG mapping flow, our framework requires two more simple processes. First, after selecting a DFG group, we route their input data that are not in the PEs assigned to their DFGs. Also, for conditionally executed DFGs, state-controlling operations (sleep instructions) are inserted at the entry of the DFGs.

After the two processes, mapping a DFG to a set of PEs can be done using known mapping algorithms (e.g., [14]–[17]). Since our framework solves the problem of handling CDFG by separating control flow and data flow, control flow needs not be considered during data flow mapping, thus conventional algorithms can be easily integrated into our framework.

V. Experiments

A. Experimental Setup

We used a CGRA named FloRA [6], [13] as the target architecture. It has an 8x8 array of PEs, and the eight PEs in each row can share the same instruction in a pipelined manner (i.e., partial SIMD mode). We implemented the CGRA with SFP in Verilog RTL and successfully synthesized it at 500MHz using TSMC 45nm technology. Its functionality was verified with gate-level simulation [9].

We extended the LLVM compiler infrastructure [18] to implement our compilation framework. Clang compiler [19] was used as the frontend tool to obtain IR. For the DFG mapping we used a variant of list-scheduling-based mapping algorithm that performs scheduling, operation-to-PE binding, and register binding all at once, similarly to [14] except that we do not adopt modulo scheduling. However, other mapping algorithms can also be used as mentioned in Section IV-D.

B. Target Applications

Any kernel having control flow can be a target application. Especially, parallel conditionals often appear as a result of loop unrolling, although they can also appear within a single iteration. Thus, we experimented with the following applications with various unrolling factors (1,2,4, and 8).

- Clipping (`clip`): it saturates values into the predefined ranges.
- Sum of absolute differences (`sad`): it calculates the sum of absolute differences between pairs of integers.
- Shift instead of division (`shift`): it divides the given integers by 16 using shift operations. If the integer is negative, then control flow is needed.

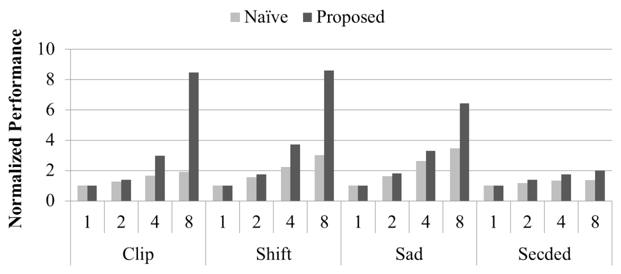


Fig. 4. Comparison of mapping results on performance. The upper X axis means the unrolling factors. The values are normalized to baselines. A baseline of each example is the case when unrolling factor is 1 and the naïve approach is used.

- SECDED decoding (`secded`): it means single-error-correction-and-double-error-detection.

C. Verification of Mapping Framework

We verified the functional correctness of the proposed mapping framework by simulating mapping results obtained from the framework on FloRA at RTL using ModelSim. We tested with total 16 cases (4 examples with 4 unrolling factors for each), and confirmed that our framework works correctly in all cases.

D. Quality of Mapping Results

We need to check our framework to see if it really exploits parallelism among multiple conditionals well. It is hard to measure the relative quality of mapping results since this is the first work for compiling applications using SFP, but one way is to compare with a naïve approach where multiple conditionals are handled sequentially. The comparison results are shown in Fig. 4. We assume that there are total 64 iterations. 8-way SIMD is supported in the architecture.

In the figure, all the examples show a similar tendency, but the results of 'sad' show what we want to do in this paper. Each iteration in 'sad' has low instruction-level parallelism so there exist many idle PEs if it executes only one iteration in a column (8 PEs). Thus, we unroll the loop to increase the utilization. However, if structures are serialized in the naïve approach, thus unrolling does not give enough benefit. On the other hand, our proposed method fully parallelizes eight *if* structures in one column, maximizing benefit from unrolling. The average improvement of our approach for the cases with unroll factor of 8 is 2.21 in the harmonic mean. Note that our work is irrelevant to how much benefit loop unrolling gives, but tries to maximize the performance given that a loop is unrolled.

VI. Conclusion

In this paper we presented a compilation framework for CGRAs that relies on SFP for conditional execution. While SFP can remove wasteful power consumption on issuing/ decoding instructions from conventional full predication, compilation becomes more challenging to handle multiple conditionals. Our technique uses a new CDFG structure that can succinctly capture the parallelism existing between conditionals, and also includes an efficient mapping algorithm for the CDFG. Especially, by separating the handling of control flow and data flow, it is also possible to be integrated with conventional mapping algorithms.

The correctness of mapping results is verified on the real architecture in RTL simulation and our experimental results demonstrate that our framework succeeded in finding and exploiting parallelism between multiple conditionals, thereby leading to 2.21 times higher performance than the naïve approach.

Acknowledgment

This work was supported in part by the National Research Foundation of Korea (NRF) funded by the Korea government (MEST) under grant No. 2012-0006272, in part by Basic Science Research Program through NRF funded by MEST under grant 2010-0011534, and in part by Ministry of Knowledge Economy and IDEC Platform center at Hanyang University.

References

[1] D. Chen and J. Rabaey, "A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high speed DSP data paths," *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 1895–1904, dec 1992.

[2] T. Miyamori and K. Olukotun, "REMARC: reconfigurable multimedia array coprocessor," *IEICE Transactions on Information and Systems*, pp. 389–397, 1998.

[3] H. Singh, M. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and M. C. Filho, "MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, pp. 465–481, may 2000.

[4] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2001.

[5] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Proceedings of the International Conference on Field Programmable Logic and Application*, 2003.

[6] D. Lee, M. Jo, K. Han, and K. Choi, "FloRA: Coarse-grained reconfigurable architecture with floating-point operation capability," in *Proceedings of the International Conference on Field-Programmable Technology*, 2009.

[7] M. L. Anido, A. Paar, and N. Bagherzadeh, "Improving the operation autonomy of SIMD processing elements by using guarded instructions and pseudo branches," in *Proceedings of the Euromicro Symposium on Digital System Design*, 2002.

[8] C. Arbelo, A. Kanstein, S. Lopez, J. Lopez, M. Berekovic, R. Sarmiento, and J. Y. Mignolet, "Mapping control-intensive video kernels onto a coarse-grain reconfigurable architecture: the H.264/AVC deblocking filter," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2007.

[9] K. Han, S. Park, and K. Choi, "State-based full predication for low power coarse-grained reconfigurable architecture," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2012.

[10] J. Shin, M. Hall, and J. Chame, "Superword-level parallelism in the presence of control flow," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2005.

[11] G. Lee, K. Chang, and K. Choi, "Automatic mapping of control-intensive kernels onto coarse-grained reconfigurable array architecture with speculative execution," in *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum*, 2010.

[12] S. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the International Symposium on Microarchitecture*, 2005.

[13] Y. Kim, M. Kiemb, C. Park, J. Jung, and K. Choi, "Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2005.

[14] B. Mei, S. Vemaldet, D. Verkestt, H. D. Man, and R. Lauwereins, "DRESC: A retargetable compiler for coarse-grained reconfigurable architectures," in *Proceedings of the International Conference on Field-Programmable Technology*, 2002.

[15] T. Toi, N. Nakamura, Y. Kato, T. Awashima, and K. Wakabayashi, "High-level synthesis challenges and solutions for a dynamically reconfigurable processor," in *Proceedings of the International Conference on Computer-Aided Design*, 2008.

[16] J. W. Yoon, A. Shrivastava, S. Park, M. Ahn, and Y. Paek, "A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architectures," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 17, pp. 1565–1578, nov 2009.

[17] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "EPIMap: using epimorphism to map applications on CGRAs," in *Proceedings of the Design Automation Conference*, 2012.

[18] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[19] http://clang.llvm.org/.