Automatic Mapping of Control-Intensive Kernels onto Coarse-Grained Reconfigurable Array Architecture with Speculative Execution

Ganghee Lee, Kyungwook Chang and Kiyoun Choi

Department of Electrical Engineering and Computer Science

Seoul National University

Seoul, South Korea

berean97@snu.ac.kr, kyungwook222@poppy.snu.ac.kr, kchoi@snu.ac.kr

Abstract— Coarse-grained reconfigurable array architectures have drawn increasing attention due to their good performance and flexibility. In general, they show high performance for compute-intensive kernel code, but cannot handle control-intensive parts efficiently, thereby degrading the overall performance. In this paper, we present automatic mapping of control-intensive kernels onto coarse-grained reconfigurable array architecture by using kernel-level speculative execution. Experimental results show that our automatic mapping tool successfully handles control-intensive kernels for coarse-grained reconfigurable array architecture. In particular, it improves the performance of the H.264 deblocking filters for luma and chroma over 26 and 16 times respectively compared to conventional software implementation. Compared to the approach using predicated execution, the proposed approach achieves 2.27 times performance enhancement.

Keywords—coarse-grained reconfigurable array architecture, mapping speculative execution, control-intensive, routing

I. INTRODUCTION

With the increasing requirements for more flexibility and higher performance in embedded systems design, reconfigurable computing is becoming more and more attractive. Although coarse-grained reconfigurable array architectures (CGRAs) can achieve performance improvement on compute-intensive kernels by exploiting the parallelism, most of the existing CGRAs have a serious limitation in handling kernels that have control-intensive characteristic. It is because of the fact that the functionality of CGRA is passively configured by the configuration controller. They cannot control the execution according to the computation results. In addition, since the processing elements (PEs) of a CGRA are tightly coupled with each other in terms of configuration, the PEs cannot be controlled independently.

These characteristics of a typical CGRA seriously limit the applicability, since there are many applications that have both control-intensive part and data-intensive part or a mixture of them. One well-known example is the deblocking filter in H.264/AVC. It basically consists of FIR filters which are heavily data-intensive, but the filtering range and coefficients are determined by many factors, resulting in control-intensive irregular data flow [2]. This control-intensive part makes it hard to map the application onto a CGRA, even though it has most time consuming data-intensive part.

To tackle this problem, various solutions have been proposed. One of them is performing control-intensive kernels with predicated executions on a CGRA [2][3]. Each PE selectively executes the delivered configurations according to its condition flag. This approach has the advantage that it can reduce the power consumption by turning off unused PEs. But it restricts the parallel execution because the condition should be checked before executing the statements inside the conditional statement.

The second approach that can be considered is running the application with speculation. Speculative execution is a way of executing all possible solutions first and then choosing one of the solutions depending on the condition. If the condition is complex and the checking takes long time, then we can execute alternative paths (threads for different conditions) concurrently while checking the condition. This solution may consume more power compared to the predicated execution, but improves the performance. Especially, in case of control-intensive kernels that have lots of common sub-expressions, this approach can be much more effective.

The problem of mapping an application onto a CGRA with a given number of resources such that the performance is maximized has been shown to be NP-complete [4]. Few automatic mapping/compiling/synthesis tools have been developed to exploit the parallelism. However, these tools are mostly oriented to map data-intensive kernels. One exception is presented in [5], where they provide a control flow analysis to replace branches with predicated operations. However, this approach is based on predicated execution, which may not give enough parallelism for the reconfigurable architecture.

In this paper, we introduce an automatic mapping of control-intensive kernels onto a CGRA with speculative execution. We apply the approach to the deblocking filter in H.264/AVC to improve the performance drastically. The automatically generated configuration gives performance results similar to those obtained by manual mapping [1].

II. TARGET ARCHITECTURE

A. Coarse-Grained Reconfigurable Array Architecture

Our target architecture consists of a reconfigurable computing module (RCM) for executing loop kernel code segments and a general purpose processor for controlling the RCM, and these units are connected with a shared bus. The RCM used in our platform consists of an array of PEs,

Figure 1. Coarse-grained reconfigurable architecture.

several sets of frame buffers, and a configuration cache memory [6]. Figure 1 shows our CGRA and internal structure of the PEs. It is connected with the nearest neighboring PEs-top, bottom, left, and right. The architecture in Figure 1 contains a 4x4 reconfigurable array of PEs, but the size of the array can be optimized to a specific application domain [6].

The PEs contain typical functional units (FUs) such as ALUs; area-critical FUs such as multipliers and dividers are located outside the PEs and shared among a set of PEs. Each area-critical FU is pipelined to curtail the critical path delay, and its execution is initiated by scheduling the area-critical operation on one of the PEs that share the area-critical FU. Thus each PE can be dynamically reconfigured either to perform arithmetic and logical operations with its own ALU in one clock cycle, or to perform multiply or division operations by using the shared FU in several clock cycles with pipelining. Pipelining the FUs improves performance by increasing the clock speed. It also helps loop pipelining execution by allowing multiple operations to execute simultaneously on one pipelined FU, thereby increasing the utilization of the FUs.

The data memory in Figure 1 is used for storing data that can be accessed by the PEs. There are two sets of memory, each of which consists of three banks: one connected to the write bus and the other two connected to the read buses. These read/write buses are also shared by the PEs like shared FUs. Two sets of memory are used for double buffering.

The configuration cache is composed of an array of Cache Elements (CEs), whose size is the same as the size of the array of PEs. More specifically, each PE has its own CE, and therefore, the two arrays (PE array and CE array) have the same dimension. Each CE has many layers, with each layer having a different context, such that the entire array of PEs can be reconfigured within just one cycle by switching the layers. Note that the area-critical resources are shared by the PEs on the same row as shown in Figure 1 and activated

Figure 2. Internal PE structure of (a) the baseline architecture and (b) the modified architecture.

through the individual PEs, and thus need not be modeled separately from the PEs.

B. Architecture Extension for Conditional Execution

To support conditional execution on the reconfigurable architecture, our target architecture has been modified slightly with an extended set of operations and additional ports [3]. Figure 2 (b) illustrates the extended reconfigurable architecture. According to the value of the newly added signal ‘Condition’, the PE can select one of the results from multiple sources (between A_{sel} and B_{sel}). The ‘Condition’ signal can be issued by the extended conditional operations (e.g., comparison or logical negation).

An interconnection network is also added for conditional execution [3]. Among various interconnect architectures, we use the ‘column-wide bus’ architecture, where a bus is placed along with each column on the array. So the total number of buses on the array is the same as the number of columns. Each bus has 1-bit width used for the ‘Condition’ signal. Note that a conditional operation should be executed just before the resulting ‘Condition’ signal is used, since in the current implementation a PE broadcasts the ‘Condition’ signal value over the ‘column-wide bus’ and it is preserved only for the next one cycle. Then the other PEs get the value from the ‘column-wide bus’ in the next cycle.

III. DESIGN FLOW AND MAPPING

Our design flow which integrates the process of mapping kernels onto the reconfigurable array is divided into three steps: code transformation, common sub-expression elimination, and kernel mapping with speculative execution.

In the code transformation step, the given plain code is transformed into the code that enables speculative execution. The conditional branching statements are removed and variables that can have different values depending on the conditions are renamed to avoid conflict. Then, at the end of each conditional statement, a statement that selects one of the results is inserted.

In the next step, common sub-expression elimination (CSE) is applied to the result of the first step. Using the CSE technique, we can extract common sub-expression from different conditional parts and the resulting sub-expression is executed only once. The speculative execution may give worse power consumption than the predicated one. However, we can eliminate many duplicated operations by applying the CSE technique. In the luma deblocking filter, for example,

TABLE I
SIMULATION RESULT OF PERFORMING DEBLOCKING FILTER ON A SUB-MACROBLOCK

	Software only	Reconfigurable architecture				Speed up		
		Manual		Automatic		SW vs. RA(automatic)	Manual vs. Automatic	
	Latency (cycles)	Latency (cycles)	Mapping time (min)	Latency (cycles)	Mapping time (min)	Latency	Latency	Mapping time
Luma	83114	3071	3360	3241	4	25.64	0.95	840
Chroma	26121	1337	1440	1601	1	16.32	0.84	1440

the number of operation nodes is reduced from 143 to 104 by applying the CSE technique. In other words, 39.27% of the nodes are eliminated. This CSE technique is a key factor to improve the efficiency of the speculative execution.

Finally, in the kernel mapping step, scheduling and binding for the kernel code that has been transformed to enable speculative execution are performed with high-level synthesis (HLS) techniques. It handles loop-level parallelism by applying loop unrolling and pipelining techniques.

Mapping a kernel onto the reconfigurable architecture requires solving multiple problems simultaneously. First, we should compile the application and generate configuration of the architecture while maximally exploiting the parallelism in both the application and the architecture. Scheduling data transfers between PEs and binding them with limited routing resources add to the complexity of the mapping process. Furthermore, mapping a transformed kernel with speculative execution is more difficult for the following reasons: i) in the application point of view, since all the condition paths are expressed in one flattened kernel code, it increases the complexity of dependency between the operations, and ii) in the architecture point of view, since the interconnect network for condition signals is added to the architecture, the search space is widened by the additional dimension of search.

To tackle this problem, we use the routing technique. When two operations (which have data or control dependency between the two operations) are mapped respectively to two different PEs that have no direct interconnection between them, we add extra dummy move operations for data forwarding [7]. Such move operations are called routing operations, and the PEs that undertake the move operations are called routing PEs. Without such routing technique, the architecture would have required shared resources such as shared registers and global buses to transfer the data. Although these shared resources simplify the mapping process, they may increase the critical path delay. In our previous design space exploration research [3], the global bus architecture which uses shared resources increases the clock period by 44 percent.

IV. EXPERIMENTS

A. Experimental Setup

The experiment is done for the deblocking filter in the H.264/AVC decoder, using a full software implementation on a RISC processor as well as an implementation on the reconfigurable architecture with speculative execution. In the software implementation, the entire deblocking filter is executed on the ARM7TDMI processor. In the

reconfigurable architecture implementation, we consider 8x8 array of PEs having mesh-plus connectivity.

The boundary strength values of a sub-block are calculated by the processor, since this process does not have any compute-intensive part, but has many conditional statements and frequent memory accesses. After the calculation of the boundary strength, we perform the deblocking filter operation on the reconfigurable architecture according to the boundary strength values. Unlike the calculation of the boundary strength, the filtering process is performed in a line by line manner. For each line in a sub-block, we should adaptively determine the filter tab and the range, and then apply the selected FIR filter on the edge of the sub-block.

The pixel values of the macroblock, which are generated from the result of the *iTrans* operation right before the deblocking process, remain in the frame buffer and are reused for the deblocking filtering. The pixel values on the left side and the upper side of the current macroblock are fetched from the external memory together with the boundary strength and quantization factor of the sub-blocks inside the current macroblock. There is communication overhead incurred due to copying this input data from SDRAM. However, the number of these pixels is not that large compared to the number of pixels in the current macroblock, thus causes rather small communication overhead.

B. Experimental Result

Table I compares the simulation results of software implementation, manual mapping, and automatically generated mapping. The communication overhead has been added to the results of reconfigurable architecture implementation for fair comparison with the software implementation. The reconfigurable architecture with automatic mapping shows 25.64 times and 16.32 times speedup in the luma deblocking and the chroma deblocking, respectively, compared to the software implementation.

It is shown in Table I that the automatically generated mapping has performance degradation of 5% and 16%, respectively, compared to the manual mapping. However, the automatic mapping reduces mapping time from hours down to minutes, thus the productivity is increased by about one thousand times. If we consider design space exploration with many iterations of optimization process, it is crucial to reduce the mapping time.

In the second experiment for exploring the architecture variations, we vary the number of PEs in a column from 4 to 8 with mesh-plus connectivity. Table II shows the mapping

Figure 3. Architecture difference.

time and performance result for each architecture instance with different number of PEs in a column. As expected, more resources give better performance.

We compare our approach using speculative execution with another approach using predicated execution on ADRES architecture [2]. It seems not easy to have a fair comparison since the architectures are quite different. Figure 3 shows the difference between the ADRES architecture and ours [6]. The two architecture instances have the same functional resources with mesh-plus connectivity. However, ADRES has several shared register files. The top 4 PEs can share data in the VLIW register file and the other PEs have their own register file but it is shared by the PEs in the diagonal positions. Our architecture does not have shared registers. Thus, communication between the PEs must be accomplished only through the bus with mesh-plus connectivity. Since our architecture does not have shared registers, it is in a disadvantageous position. In [2], for instance, they say that it achieves 29% performance enhancement by using shared register files. Shared resources may give better performance in terms of number of clock cycles. However, as we have mentioned in Section III, it may decrease the clock speed.

Table III shows performance comparison between the result of [2] and that of ours. In [2], the worst-case execution times of deblocking filter in H.264/AVC using predicated execution are 29,799,770 cycles and 11,797,104 cycles for luma block and chroma block respectively. Since they use CIF image which has 396 macroblocks, it takes 105,043 cycles per macroblock (a macroblock contains four 8x8 pixel luma images and two 8x8 pixel chroma images). Our result using speculative execution takes 24,358 cycles per macroblock. Even though we don't have any shared register files, it achieves 4.31 times higher performance. In the average case, predicated execution in [2] takes 55,203 cycles per macroblock assuming the same probability between weak filtering and strong filtering. Since we use speculative execution technique, we always have the same performance result, thereby achieve 2.27 times performance improvement.

V. CONCLUSION

In this paper, we presented an automatic mapping of control-intensive kernels through kernel-level speculative execution on a CGRA. For the speculative execution, the

TABLE II
SIMULATION RESULT FOR DIFFERENT NUMBERS OF PEs IN A COLUMN

	4 PEs in a column		8 PEs in a column	
	Latency (cycles)	Mapping time (min)	Latency (cycles)	Mapping time (min)
Luma	5097	7	3241	4
Chroma	1985	1	1601	1

TABLE III
PERFORMANCE COMPARISON

	Predicated execution [2]	Speculative execution	Speed up
Worst-case	105,043	24,358	4.31
Average	55,203	24,358	2.27

array of PEs executes alternative paths concurrently and then one of the results is selected according to the condition. With this technique, one can extract parallelism from the application and utilize unused PEs in the CGRA. The experiment with the deblocking filter of H.264/AVC shows 26 and 16 times performance improvement for a luma macroblock and a chroma macroblock, respectively, compared to software implementation, and 5% and 16% lower performance, respectively, compared to manual mapping. Compared to the approach using predicated execution, the proposed approach achieves 2.27 times higher performance.

Using the automatic mapping tool, the designer can map control-intensive kernel code as well as data-intensive kernel code without knowing the internal details of complex CGRAs. And the designer will also be relieved from tedious and error-prone tasks of manual mapping.

ACKNOWLEDGMENT

This work was supported by KOSEF under NRL Program Grant (R0A-2008-000-20126-0) funded by MEST and IITA under ITRC support program (IITA-2008-C1090-0804-0009) funded by MKE.

REFERENCES

- [1] K. Chang and K. Choi, "Mapping control-intensive kernels onto coarse-grained reconfigurable array architecture," in *Proc. ISOCC*, 2008.
- [2] C. Albello, A. Kanstein, S. Lopez, J.F. Lopez, M. Berekovic, R. Sarmiento, and J.Y. Mignolet "Mapping control-intensive video kernels onto a coarse-grained reconfigurable architecture: the H.264/AVC deblocking filter," in *Proc. DATE*, 2007.
- [3] J. Lee, Y. Kim, J. Jung, S. Kang, K. Choi, "Reconfigurable ALU array architecture with conditional execution," in *Proc. ISOCC*, 2004.
- [4] C. Oliver Shields, Jr., *Area Efficient Layouts of Binary Trees in Grids*, Ph.D thesis, the University of Texas at Dallas, USA, 2001.
- [5] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "DRESC: A retargetable compiler for coarse-grained reconfigurable architectures," in *Proc. ICFPT*, 2002.
- [6] 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 *Prom. DATE*, 2005.
- [7] G. Lee, S. Lee, K. Choi, and N. Dutt, "Routing-aware application mapping considering Steiner points for coarse-grained reconfigurable architecture," in *Proc. ARC*, 2010.