Improving Performance of Nested Loops on Reconfigurable Array Processors

YONGJOO KIM, Seoul National University
JONGEUN LEE and TOAN X. MAI, UNIST
YUNHEUNG PAEK, Seoul National University

Pipelining algorithms are typically concerned with improving only the steady-state performance, or the kernel time. The pipeline setup time happens only once and therefore can be negligible compared to the kernel time. However, for Coarse-Grained Reconfigurable Architectures (CGRAs) used as a coprocessor to a main processor, pipeline setup can take much longer due to the communication delay between the two processors, and can become significant if it is repeated in an outer loop of a loop nest. In this paper we evaluate the overhead of such non-kernel execution times when mapping nested loops for CGRAs, and propose a novel architecture-compiler cooperative scheme to reduce the overhead, while also minimizing the number of extra configurations required. Our experimental results using loops from multimedia and scientific domains demonstrate that our proposed techniques can greatly increase the performance of nested loops by up to 2.87 times compared to the conventional approach of accelerating only the innermost loops. Moreover, the mappings generated by our techniques require only a modest number of configurations that can fit in recent reconfigurable architectures.

Categories and Subject Descriptors: C.3 [Special-Purpose and Application-Based Systems]: *Real-time and embedded systems*; D.3.4 [Programming Languages]: *Processors—Code generation; optimization*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Coarse-grained reconfigurable architecture, compilation, nested loop, software pipelining

ACM Reference Format:

Kim, Y., Lee, J., Mai, T. X., and Paek, Y. 2012. Improving performance of nested loops on reconfigurable array processors. *ACM Trans. Architect. Code Optim.* 8, 4, Article 32 (January 2012), 23 pages.
DOI = 10.1145/2086696.2086711 <http://doi.acm.org/10.1145/2086696.2086711>

1. INTRODUCTION

With the increasing interest in many-core architectures, partly driven by continuing transistor scaling and consequently the expected exponential growth in the number of on-chip cores [Patterson and Hennessy 2008], the importance of how to make the most of the available on-chip cores by programs is rising. This problem is more pronounced in

This work was supported in part by the Korea Science and Engineering Foundation(KOSEF) NRL Program grant funded by the Korea government (MEST) (No. 2011-0018609), the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST)/Korea Science and Engineering Foundation (KOSEF) (Grant 2011-0000975), IDEC, and in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by MEST, under grant 2010-0011534.

Authors' addresses: Y. Kim and Y. Paek, The Software Optimizations and Restructuring Group, School of Electrical Engineering and Computer Sciences, Seoul National University, Shinlim-dong, Gwanak-gu, Seoul, 151-744, Korea; J. Lee and T. Mai, Energy-Efficient Computing Lab, School of ECE, UNIST, 100 Banyeon-ri, Ulsan 689-798, South Korea; J. Lee (corresponding author), email: jlee@unist.ac.kr.

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.

© 2012 ACM 1544-3566/2012/01-ART32 \$10.00

DOI 10.1145/2086696.2086711 <http://doi.acm.org/10.1145/2086696.2086711>

the case of Coarse-Grained Reconfigurable Architectures (CGRAs) [Hartenstein 2001], where each Processing Element (PE) is much simpler and there are much more of them on a single processor, than in a typical multi-core processor. Pipelining is the most widely used way of mapping loops onto CGRAs [Mei et al. 2003; Park et al. 2008]. With pipelining, the operations of a loop are mapped to different PEs, such that each PE will perform only the operation mapped to it for the entire iterations of a loop, thereby achieving the Initiation Interval (II) of one cycle, or in other words, initiating a new iteration every cycle.¹ Thus the steady-state performance of CGRAs can be many times higher than single- or multi-core processors.

However, there are certain things that must be done by a main processor, to which a CGRA is usually used as a coprocessor or an accelerator. The main processor typically takes care of sequential code execution as well as managing the CGRA in terms of input/output data preparation and pipeline setup. Although these things are one-time jobs and therefore account for much less execution cycles than loop kernel executions, they too can be significant if (i) they are repeated and (ii) they involve communication between the main processor and the CGRA. Such is the case when only the innermost loop of a loop nest is accelerated by CGRAs using today's state-of-the-art mapping approaches. We evaluate in this paper the overhead of sequential computation in nested loops taking into account the detailed communication between main processor and CGRA coprocessor, and propose a novel hardware-compiler cooperative scheme to drastically reduce the overhead.

One effective way to reduce the impact of sequential computation in nested loops is to pipeline outer loops as well. Though outer-loop pipelining techniques have been proposed for VLIW processors [Rong et al. 2004; Muthukumar and Doshi 2001], it is very difficult to apply them directly to CGRAs due to important architectural differences between VLIWs and CGRAs, such as CGRAs' lack of central register files and dependence on main processor for sequential computation as well as pipeline setup. Such limitations can create problems with, for instance, rescheduling sequential code and inner-loop codes, thus significantly degrading the quality of code of outer-loop pipelining. To avoid these problems, we propose a minimal hardware extension and compiler optimization that enable efficient pipelining of nested loops up to two levels. Another issue in pipelining outer loops for CGRAs is to minimize the *number of configurations*, which is analogous to, and even more critical than, code size for VLIWs, due to the limited number of configurations that can be kept locally. Our scheme uses only a small number of extra configurations through (i) a novel sequential execution architecture (Section 5.2.2) and (ii) a careful epilog-prolog merging (Section 5.3).

One practical challenge in evaluating our proposed scheme is that the amount of communication overhead due to pipeline setup (for the innermost loop) heavily depends on the implementation details that are often hard to find in the literature, such as how pipeline control is done at the cycle level, how many parameters are needed for pipeline control, and how those parameters are passed to a CGRA through instructions. In this paper we present a detailed model of single-loop pipelining including a pipeline control algorithm and the software interface (Section 3).

Our experimental results using important loops from multimedia and scientific domains demonstrate that our nested-loop pipelining techniques can greatly improve nested loop execution performance by up to 2.87 times, or 2.08 times on average, compared to the conventional approach of accelerating only the innermost loops. Moreover, the mappings generated by our techniques require only a modest number of

¹If there are more operations than can be mapped to the available PEs, multiple ($= N$) operations must be mapped to each PE, leading to the initiation interval of N cycles.

configurations, which are well within the range of reconfiguration capacity of recent reconfigurable architectures.

The rest of the paper is organized as follows. In Section 2 we discuss the related work. In Section 3 we present single-loop pipelining for CGRAs, based on which we motivate the need for nested-loop pipelining in Section 4. In Section 5 we present nested-loop pipelining. We discuss our experimental results in Section 6 and conclude the paper in Section 7.

2. RELATED WORK

Pipelining loops has been studied extensively for various architectures. ASICs (Application-Specific Integrated Circuits) and FPGA designs, especially in the domain of digital signal processing, primarily capitalize on pipelining [Petkov et al. 2002; Bondalapati 2001] to increase the throughput of the system, which typically performs the same function a great number of times.

For VLIW processors, successive loop iterations can be overlapped to maximize instruction level parallelism, called software pipelining [Rau 1994]. Pipelining of nested loops for VLIW processors is considered by several approaches [Wang and Gao 1996; Muthukumar and Doshi 2001; Rong et al. 2004; Fellahi and Cohen 2009]. Wang and Gao [1996] propose pipelining-dovetailing, which targets perfectly nested loops and extends the software pipelining of the innermost loop to the surrounding loop nests by dovetailing successive outer-loop iterations. Similarly Muthukumar and Doshi [2001] propose a technique to overlap multiple inner-loop pipelines to achieve greater parallelism. Rong et al. [2004], rather than extending modulo scheduling from the innermost loop toward outer loops, propose a method to directly apply software pipelining at an arbitrary loop level, thereby increasing the freedom of scheduling. For loop nests with a series of inner loops, or “phases”, Fellahi and Cohen [2009] propose prolog-epilog merging between successive phases. However, unlike VLIW processors where both loops and sequential codes are executed on the same core, CGRAs can execute loops only and sequential codes including even pipeline setup must be done by a main processor, which makes it impossible to directly apply the above techniques to CGRAs. Moreover, all the above compilation techniques are very difficult to implement for reconfigurable architectures, which typically lack central register files and branch/loop-back instructions.

CGRAs saw the application of modulo scheduling only recently. Mei et al. [2003] is the first approach to apply modulo scheduling to CGRAs with 2D array of processing elements, though Callahan and Wawrzynek [2000] earlier applied modulo scheduling to their 1D reconfigurable architecture. Callahan and Wawrzynek [2000] also presented automatic generation of prolog and epilog at runtime, based on their specialized control hardware. Most mapping algorithms for CGRAs are limited to innermost loops only [Mei et al. 2002; Kim et al. 2005; Park et al. 2008], though they could be applied to an outer loop if all the inner loops were fully unrolled first. Recently Park et al. [2009a] consider executing multiple unrelated pipelines simultaneously on the same CGRA, exploiting task-level parallelism for streaming applications.

Certain loop nests easily lend themselves to polytope-based loop parallelization, which is applied to CGRAs [Hannig et al. 2004; Dutta et al. 2009] as well as VLSI designs and systolic arrays. This method can handle arbitrary loop levels and provide a framework to generate highly efficient implementations taking into account the architectural constraints. However, it is not known how to automatically generate such implementations for an arbitrary loop nest.

The problem of mapping sequential code onto CGRAs has received little attention. Park et al. [2009b] address this problem with operation chaining enabled by bypass networks, which may require significant architecture modification. Our focus

Fig. 1. CGRA architecture.

is on minimizing configuration size rather than performance maximization, with little change in the architecture except for configuration sequencing.

3. SINGLE-LOOP PIPELINING FOR CGRA

3.1. CGRA Architecture

CGRA, illustrated in Figure 1, is based on an array of Processing Elements (PEs), each of which can perform a set of arithmetic operations. Similarly to fine-grained reconfigurable architectures such as FPGAs, the operation of each PE and interconnections can be configured via distributed context registers, or *configurations*, but unlike FPGAs, each PE performs word-level operations instead of bit-level operations. This coarser granularity enables smaller configurations, faster reconfiguration time, and more efficient implementation of compute operations. Each PE has its own small set of registers for constants and temporary variables. In addition, a set of rotating predicate (1-bit) registers is shared among all PEs, which allows for easier handling of conditionals within loops. Array variables are stored in the local memory of a CGRA, which is organized as a multi-bank, multi-port on-chip memory. The local memory can be directly accessed by some PEs. A CGRA is used as an accelerator to main processor, and executions of the two processors are exclusive to each other. In other words, main processor does not perform any useful work while CGRA is in execution. However, configurations and input data can be pre-loaded into CGRA through DMA (Direct Memory Access), which may overlap with main processor executions.

3.2. Single-Loop Pipelining

Let us consider, for a CGRA illustrated in Figure 2(a), how to map and execute a loop whose body has operations represented by the Data Flow Graph (DFG) in Figure 2(b). For simpler illustration the loop's trip count is set to 4, and a PE is assumed to execute any operation in one cycle. One mapping of the DFG to the PE array is given in Figure 2(c), which, without pipelining, will result in 4 cycles per iteration. With pipelining, the II (Initiation Interval) of one cycle can be achieved, as illustrated in Figure 2(e). However, pipelining requires more complicated control.

3.2.1. Automatic Prolog/Epilog Generation. One complexity in pipelining is how to correctly handle prolog and epilog, which are the initial and the final iterations in which a pipeline is not full. Prolog/epilog code could be generated separately from the kernel

0 _(p0)	3 _(p2)	4 _(p3)
1 _(p0)	2 _(p1)	x

(d) Configuration for pipelined execution, which has four stages, denoted by subscripts, p0 ~ p3.

	Cycle	PE arrays			Predicate Register				Control Unit	
					p0	p1	p2	p3	LC	EC
Prolog	1	0 _(1st)	x	x	1	0	0	0	3	4
		1 _(1st)	x	x						
	2	0 _(2nd)	x	x	1	1	0	0	2	4
Kernel		1 _(2nd)	2 _(1st)	x						
	3	0 _(3rd)	3 _(1st)	x	1	1	1	0	1	4
		1 _(3rd)	2 _(2nd)	x						
Epilog	4	0 _(4th)	3 _(2nd)	4 _(1st)	1	1	1	1	0	4
		1 _(4th)	2 _(3rd)	x						
	5	x	3 _(3rd)	4 _(2nd)	0	1	1	1	0	3
		x	2 _(4th)	x						
	6	x	3 _(4th)	4 _(3rd)	0	0	1	1	0	2
		x	x	x						
	7	x	x	4 _(4th)	0	0	0	1	0	1
		x	x	x						

(e) Loop execution with pipelining (LC: Loop Count, EC: Epilog Count), where a subscript represents the iteration for which computation is done. For instance, at cycle 1, PE0 performs the operation of node 0 for the 1st iteration.

Fig. 2. Pipelining for CGRA.

code during compilation, which however is very likely to increase the code size or the number of configurations, and therefore detrimental for CGRAs, as CGRAs can keep only a very limited number of configurations locally.² If instead prolog/epilog configurations can be generated from the kernel configurations at runtime by the CGRA itself, no extra configuration will be necessary. To generate prolog/epilog configurations automatically at runtime, we borrow a similar concept from the VLIW software pipelining community, which is based on Rotating Predicate Registers (RPRs) [Huck et al. 2000].

RPRs, denoted by $p_0, p_1, \dots$, are very similar to a shift register, capable of simultaneously shifting the value of p_0 to p_1 , that of p_1 to p_2 , and so on. The shift operation happens every II cycles. Thus the CGRA control unit must be aware of the II parameter. The other parameters of which the CGRA control unit needs to be aware include LC (Loop Count), which is the trip count of the loop, and EC (Epilog Count), which is the schedule length in terms of stages.

Algorithm 1 illustrates how to correctly generate prolog and epilog at runtime using those registers, where SCN (Start Configuration Number) is the start address of the configurations that will be used. The configuration for our pipelining example is given in Figure 2(d), where the subscripts represent RPRs associated with each part of the configuration. The RPRs can be generated as follows: the part of the configuration that corresponds to a PE scheduled to step i is predicated by p_{i-1} . Thus, as per the algorithm, when the pipeline control starts writing ‘1’ to p_0 , it enables only the part of the configuration that corresponds to the first stage of the prolog (see Figure 2(e)). Then, after every II cycles, more of the configuration becomes enabled, until the kernel is reached. Similarly, after the kernel is finished, epilog configurations can be automatically generated by writing zeros to p_0 . This algorithm is part of the CGRA control unit, and can be implemented in hardware using FSM (Finite-State Machine).³

ALGORITHM 1: Execute-Pipeline (II , LC , EC , SCN)

```

1: Clear RPRs                                     /* RPR: Rotating Predicate Register */
2: loop
3:   if  $LC > 0$  then
4:     Rotate RPRs, writing 1 to  $p_0$ 
5:      $LC \leftarrow LC - 1$ 
6:   else if  $EC > 0$  then
7:     Rotate RPRs, writing 0 to  $p_0$ 
8:      $EC \leftarrow EC - 1$ 
9:   else                                           /*  $LC = 0$ ,  $EC = 0$  */
10:    break                                       /* exit the pipeline */
11:  end if
12:  Execute CGRA for  $II$  cycles using  $II$  configs at  $SCN$ 
13: end loop

```

3.2.2. Software Interface. Table I summarizes the main processor instructions related to CGRA. `C.Run` starts pipeline execution; for instance, this instruction can initiate an FSM implementing Algorithm 1. Parameters for pipeline execution may be set by another instruction, `C.SetupPipe`, which merely stores into CGRA control registers the pipeline-related parameters such as II , LC , EC , and SCN . `C.LoadConf` brings in necessary configurations from the configuration memory to the distributed configuration

²For instance, MorphoSys [Singh et al. 2000] has only 32 entries in the configuration cache.

³While-loops can be implemented by evaluating conditions in PEs and enabling PEs to generate an event (e.g., `StopLoop`) that is collected (OR-ed) by the CGRA control unit.

Table I. Instructions for Single-Loop Pipelining

Coproc Instruction	Parameters	Operation
C_Run	None	Execute pipeline, generating prolog/epilog automatically
C_SetupPipe	II, LC, EC, SCN	Transfer parameters from main processor to coprocessor
C_LoadConf	SCN, N, BaseAddr	Load N configurations from config memory to config cache

Table II. Loop Execution Scenario (many steps are optional)

Main Processor Execution	CGRA Coprocessor Operation
C_LoadConf	Config cache loading
C_LoadArray	DMA operation
Loop-invariant code	None
Simple data transfer (e.g., base address)	PE register access
C_SetupPipe	Control register access
C_Run	Prolog execution Kernel execution Epilog execution
C_StoreArray	DMA operation
Simple data transfer	PE register access

Fig. 3. System architecture with a CGRA.

cache. This operation may take hundreds of cycles, which may be hidden by issuing the instruction in advance. In addition, though not strictly related to pipelining, data transfer instructions (e.g., C_LoadArray, C_StoreArray) can be used to transfer data between main processor and CGRA coprocessor via DMA. Note that special instructions though they may seem, these instructions can be implemented as memory-mapped I/O using generic load/store instructions. For simple data transfers (e.g., sending a few words), we assume that the main processor can access the distributed PE-registers through memory-mapped I/O while the CGRA is not in execution.

4. MOTIVATING EXAMPLE

Executing a loop involves various “initializations” before pipeline can begin. This is true even when pipeline execution happens on the same processor as the rest of the code (e.g., VLIW processor) because of loop-invariant code (e.g., initialization of pointer’s base address). However, more needs to be done on a hybrid architecture consisting of a main processor and a CGRA coprocessor, as illustrated in Figure 3.

Table II lists the steps of a loop execution procedure for a CGRA coprocessor. In addition to executing some loop-invariant code and starting the pipeline with C_Run, main processor must transfer pipeline setup parameters to CGRA. Also, it may need to transfer other parameters such as base addresses of input arrays, even if we assume

Table III. Analysis of FIR Loop Execution Time

	Main Processor		CGRA	
	Cycle	%	Cycle	%
Pre-innerloop statement	935	12.2		
Loop-invariant code	220	2.9		
Simple data transfer	1485	19.3		
C.Run	440	5.7		
CGRA execution			3300	42.9
Interrupt from CGRA	440	5.7		
Outerloop condition check	825	10.7		
Misc.	44	0.6		
Total	4389	57.1	3300	42.9

that large data can be transferred in advance through DMA. The main processor code for these operations is called *preheader* [Muchnick 1997]. The existence of preheader code implies that even if the loop is perfectly nested by an outer loop, the pipelined execution of the inner loop, which will be repeated as many times as the number of outer loop iterations, cannot be joined together in time. The gaps can be considerable for CGRA coprocessors due to the communication latency between the two processors.

To assess how much the overhead can be, we use a simple FIR (Finite Impulse Response) filter example, which is extended to a doubly nested loop, with the input/output arrays also extended from 1D to 2D. The inner loop's trip count as well as the outer loop's is 10, and the loop nest is imperfectly nested, viz., there are a few statements (for iterator initialization) before the inner loop.

Table III summarizes our execution time analysis on the loop nest in terms of main processor cycles, as executed on the target architecture illustrated in Figure 3 (see Section 6.1 for details of our evaluation methodology). Here we assume that the pipeline setup parameters are reused except for the first iteration of the outer loop, and that the configuration load and array load operations can be done early enough for the entire loop nest, so that their delay can be fully hidden by other main processor code execution.

This table reveals that the CGRA runtime accounts for only 42.9% of the total execution time. Setting up the pipeline and getting back to main processor accounts for 30.7% ($=19.3 + 5.7 + 5.7$). Also significant is the rest of the computation such as the initialization and looping control for the outer loop, for which abundant CGRA resources are of little help. This example illustrates the significance of the overhead due to repeated communication between main processor and CGRA, which can be nearly eliminated by pipelining the outer loop. In the next section we discuss how to extend architecture and compiler to achieve this.

5. NESTED LOOP PIPELINING

Single-loop pipelining can generate a pipeline for the innermost loop only. Therefore single-loop pipelining is also Inner-Loop Pipelining (ILP).⁴ For CGRAs the conventional approach to loop nests has been inner-loop pipelining with the outer loop implemented in software on a main processor, which we refer to as *ILP + SW Outerloop*.

This approach not only suffers from significant performance overhead due to communication in hybrid architectures, but it is also unable to take advantage of outer-loop pipelining. We address those problems by (i) delegating the entire loop nest to CGRA up to two levels, which we call *HW Outerloop*, and (ii) overlapping outer-loop iterations through sophisticated rescheduling on CGRA, called *Outerloop Pipelining*. Note that mapping the entire loop nest onto CGRA can not only reduce communication delay, but

⁴In this paper the acronym "ILP" is used exclusively for Inner-Loop Pipelining.

Fig. 4. Inner-loop pipeline.

	<u>Baseline</u>	<u>Innerloop Pipeline + HW Outerloop</u>
Outerloop-preheader	Software	Software
Pre-innerloop Statements	Software	Configuration
Innerloop Pipeline	Software + Configuration	Configuration
Post-innerloop Statements	Software	Configuration
Iterator Update	Software	Configuration
Condition Check / Loopback	Software	Hardware

Fig. 5. Outer-loop execution: software vs. hardware. Here “configuration” means that the corresponding steps can be done by programming the reconfigurable architecture appropriately.

also eliminate the need for communication at all. The combination of the two is called *Nested-Loop Pipelining (NLP)*.

We target nested loops containing only a single inner loop (as opposed to multiple inner loops in a series), including both perfect and imperfect loop nests, as this type of loops account for 97% of nested loops in our analysis of a representative multimedia application (H.264 decoder [H.264]).

To help understand our NLP technique let us consider the following three cases.

- Case 1 (*baseline*). ILP + SW Outerloop
- Case 2 (*intermediate*). ILP + HW Outerloop
- Case 3 (*NLP*). Case 2 + Outerloop Pipelining

5.1. Baseline: ILP + SW Outerloop

For our discussion of nested loop executions, an inner-loop pipeline can be viewed as a four-step process consisting of preheader, prolog, kernel, and epilog (see Figure 4). This abstraction originates from the more detailed loop execution diagram in Table II. The preheader, which includes loop-invariant code execution as well as issuing some pipeline-related instructions such as `C.PipeSetup` and `C.Run`, is performed primarily by the main processor, which may enlist the help of CGRA. However, the other steps, viz., prolog, kernel, and epilog, are performed autonomously by the CGRA coprocessor.

Now if the inner-loop pipeline is enclosed by another loop, the baseline approach implements the outer loop in software on the main processor. What is added by the outer loop is, as illustrated in Figure 5, iterator update, condition check, and loop back, plus extra code between them such as outer-loop preheader, and pre-innerloop and post-innerloop statements for imperfectly nested loops. Again, all of these are executed

		0 _(p0)	3 _(p2)	4 _(p3)	
		1 _(p0)	2 _(p1)	x	

(a) Sequential code configuration

Cycle	PE Array			Predicate Register				Cont. Unit
				p0	p1	p2	p3	SC
1	0	x	x	1	0	0	0	4
	1	x	x					
2	x	x	x	0	1	0	0	3
	x	2	x					
3	x	3	x	0	0	1	0	2
	x	x	x					
4	x	x	4	0	0	0	1	1
	x	x	x					

(b) Execution

(b) Execution

Fig. 6. Example of sequential execution (SC: Stage Count).

in software on the main processor, and only the inner-loop pipeline is executed on the CGRA coprocessor,⁵ which means that there can be a significant amount of communication overhead for every outer-loop iteration.

5.2. ILP + HW Outerloop

To eliminate the communication overhead present in the baseline case, HW Outerloop extends the CGRA's pipeline control so that a two-level nested loop can be directly supported. In this case, the sequence of operations is exactly the same as in the baseline case, but only where they are executed is different (see Figure 5). Most of the software code is translated and executed now on the CGRA coprocessor, while condition check and loop back operations may be incorporated into the CGRA control unit itself. Two changes are required in the CGRA architecture.

5.2.1. Outer Loop Control. First, the part of the CGRA control unit responsible for pipeline control, such as FSM and control registers, has to be extended for outer-loop looping as well as newly added sequential codes (or configurations, after translated). Implementing outer-loop looping is straightforward, which can be done by adding LC2 (outerloop trip count) register and adding a few more states in the FSM. The four new sequential configurations can be executed by CGRA once the CGRA control unit is aware of their existence and parameters.

5.2.2. Sequential Code Execution Using CGRA. Now the second change is related to minimizing the number of configurations in mapping sequential code. The question of how to best map and execute sequential code using CGRAs has received very little attention so far. Sequential code, due to its very nature, lacks in instruction level parallelism. Therefore if we map sequential code onto CGRA, it is very likely that only a few PEs can be utilized per cycle, which means that most of the words in a configuration "bundle" will be wasted. Instead, we can maximize reuse of configurations by executing different parts of a configuration every cycle using the predication mechanism, as illustrated in Figure 6. In this example, executing the sequential code requires four cycles if enough resources are provided; thus, the naïve scheme would generate and use four configurations for four cycles. However, our scheme uses only one configuration for the

⁵The preheader of an inner-loop pipeline involves the main processor.

same number of cycles. In short we greatly reduce configuration size by compressing the active portions of four configurations into one.

For longer chains of sequential operations, more than one configurations may be necessary. In this case we use multiple configurations in a round-robin manner, much the same way as multiple configurations are used in kernel execution. This is simpler than alternatives, and can be parameterized with just 3 numbers: N (number of configurations), SC (Stage Count), and SCN (Start Configuration Number), as illustrated in Algorithm 2. An alternative is to finish up one configuration before using the next one. In this case, however, stage count (i.e., schedule length) for each configuration might be different, thus requiring more, and even a variable number of parameters for the control FSM to capture the execution plan. In addition, our approach makes it easier to generate sequential code, which can be done similarly to iterative modulo scheduling [Rau 1994].

ALGORITHM 2: Execute-Sequential(N , SC , SCN)

```

1: Clear RPRs                                     /* RPR: Rotating Predicate Register */
2: Set  $p_0 \leftarrow 1$ 
3: while  $SC > 0$  do
4:   Rotate RPRs, writing 0 to  $p_0$ 
5:    $SC \leftarrow SC - 1$ 
6:   Execute CGRA for  $N$  cycles using  $N$  configs at  $SCN$ 
7: end while

```

In summary, the benefit of HW Outerloop is three-fold. First, the communication overhead is eliminated as the entire loop nest is executed on a CGRA autonomously. Second, sequential codes may run faster as they now run on a CGRA, which is usually more resourceful than the main processor. Third and most importantly, HW Outerloop opens up the possibility of pipelining outer-loop iterations, which is now explained.

5.3. Outerloop Pipelining

5.3.1. Overview. Figure 7, which combines Figures 4 and 5, shows a close-up view of a nested-loop execution on a CGRA using HW Outerloop. In general there are up to four types of sequential configurations that can come between two consecutive inner-loop pipeline executions: post-innerloop statements, iterator update, pre-innerloop statements, and innerloop preheader. Let us refer to them simply as sequential configurations. On a higher level, nested-loop execution with HW Outerloop can be seen as a repetition of sequential, prolog, kernel, and epilog configurations.

First we observe that the nested-loop execution in Figure 7 can be easily compressed if there were no sequential configurations at all between every pair of epilog and the following prolog. Second, if epilog and prolog configurations are merged together, the merged one is exactly identical to the kernel configuration, because prolog and epilog configurations are the exact mirrors of each other. (Recall that writing ones to p_0 generates prolog configurations while writing zeros generates epilog configurations.) Therefore, nested-loop pipelining could be achieved by simply executing inner-loop kernels many times.

Driven by this insight, our strategy is to divide up the sequential configurations and see if we can reschedule them away from the epilog-prolog pair. If we can divide them into these two partitions, namely, Epilog-Independent Configurations (EIC) and Prolog-Independent Configurations (PIC), we can dissolve them by moving EIC up above the previous epilog and PIC down below the next prolog. Consequently, the remaining epilog-prolog pair (an epilog and the next prolog) can be merged into what is identical

Fig. 7. Rescheduling sequential configurations in Case 2 (EIC: Epilog-Independent Configuration, PIC: Prolog-Independent Configuration).

Fig. 8. Nested-loop pipelining (see Figure 7 for the meaning of EIC and PIC).

to kernel. Note that in this case sequential configurations are there, but because they are rescheduled to somewhere else, we can merge epilog and prolog seamlessly.

Figure 8 illustrates the execution of nested-loop pipelining after merging, which can be divided into three parts. The first part is from the beginning up to the first prolog,

which can be called *outerloop prolog*. The second part is from the first kernel up to the last PIC, right before the last kernel. This part is periodic, consisting of $NC2 - 1$ repetitions of the kernel-EIC-kernel-PIC pattern, (where $NC2$ is the outer loop's trip count) and may be called *outerloop kernel*. The rest can be called *outerloop epilog*.

The CGRA can autonomously execute the outerloop kernel plus the first prolog and the last kernel and epilog. The rest can be done in software on the main processor. In the remainder we discuss three important issues of nested-loop pipelining.

5.3.2. Finding EIC and PIC. For every pair of an epilog and the next prolog (i.e., the prolog of the next outer-loop iteration), there may exist a number of sequential configurations, which we try partitioning into EIC and PIC. If successful, EIC and PIC are rescheduled so that the pair can be merged seamlessly; otherwise, we cannot apply outerloop pipelining to that loop nest.

We use Data Flow Graph (DFG) representation of the sequential configurations, which often exists as a set of disconnected DFGs rather than a single connected one. We deal with each connected DFG as a single element in our partitioning. For example, we check if an entire DFG has no flow dependency on the epilog, and if so, it can be moved past the epilog. Even if the DFG has only a single node that is dependent on the epilog, we do not attempt to break the DFG. This is because breaking a DFG in this manner must result in a data flow edge hanging over a kernel period,⁶ creating a long data flow to realize, which usually requires many (rotating) registers over multiple cycles. If an entire DFG is moved, no additional register is required. Moreover, in perfect loops quite often the entire sequential configurations qualify as EIC.

Partitioning itself is very simple (see Algorithm 3). EIC is the set of DFGs that do not have flow dependency on the previous epilog.⁷ PIC is the set of DFGs that do have flow dependency on the previous epilog but on which the next prolog does not have flow dependency. If there is any DFG that have flow dependency on the previous epilog and on which the next prolog also has flow dependency, epilog-prolog merging cannot happen for the loop nest.

ALGORITHM 3: Find-EIC-PIC(*epilog*, *prolog*, *set-of-DFGs*)

```

1: Clear EIC and PIC
2: for each DFG d in set-of-DFGs do
3:   if d does not depend on epilog then
4:     Add d to EIC
5:   else if prolog does not depend on d then
6:     Add d to PIC
7:   else                                     /* d has dependency in both directions */
8:     Fail
9:   end if
10: end for

```

After finding EIC and PIC, the DFGs in each group are scheduled onto CGRA using the method described in Section 5.2.2.

5.3.3. Anti- and Output Dependencies. The EIC-PIC partitioning is concerned with flow dependency only. Other types of data dependencies, namely anti- and output

⁶This kernel period is due to the merging of epilog and prolog.

⁷In practice, we determine EIC and PIC before scheduling, or before knowing what exactly prolog and epilog are. This is so that we can schedule EIC before the loop body, which is much easier than the other way due to less scheduling constraints. We conservatively assume that the entire loop body is epilog (or prolog) for dependency check.

Fig. 9. Compilation example. Grey nodes in DFGs are operations that access local registers.

dependencies, can cause a problem, which however can be handled with a mechanism that is essentially renaming.

Consider a loop nest whose inner loop has a loop body as shown in Figure 9(a), and whose outer loop has index initialization code as shown in Figure 9(b) above the inner loop. Note that the loop body contains $i \leftarrow i + 1$ while index initialization includes $i \leftarrow 0$. Mapping the inner loop onto the PE array as shown in Figure 9(c) generates a mapping, which consists of two configurations (thus, Π is 2) as shown in Figure 9(d). Since the schedule length is 2 stages (1 stage = 2 cycles), two predicate registers are needed.

Figure 9(e) illustrates how the loop nest is executed on a CGRA, without Outerloop Pipelining. The first two cycles show the kernel of the innerloop, and the next two cycles are the epilog, which is followed by index initialization for the next outerloop iteration.

The index initialization code does not have flow dependency on the epilog, since the value of variable i , for instance, is defined in the epilog, but is not used in the index initialization code. There are only anti- and output dependencies. Thus the index initialization qualifies as EIC, and we would move it to above epilog. However if we do, the semantics is changed, and i gets a wrong value as illustrated in Figure 9(f).

We solve this problem by changing the way the (inner) loop body is scheduled. To state the problem, EIC has no flow dependency on the preceding epilog, but may have anti- or output dependency on it. We would like to move EIC to right before the epilog without violating anti- or output dependency. Let i be a variable with anti- or output dependency; that is, i is read or written in the epilog, and written in EIC. A solution is to rename the variable i in the epilog. A “write” variable is easy to rename but to rename a “read” variable we must still read the variable at least once—which we can do in the first stage. This doesn’t create a dependency problem because the first stage is not included, by definition, in epilog. Thus we avoid the problem by adding *stage constraint* for variables with anti-dependency, that they have to be read in the first stage (and kept, if necessary, for subsequent accesses in the loop body). Likewise, we schedule “writes” to variables on which PIC has anti- or output dependency, to the last stage. This works whether the variable is a scalar or an array element.

In our example, node 3 is originally mapped to stage 2 (p_1) though it has anti-dependency, thus resulting in incorrect behavior when EIC is moved. In a new mapping illustrated in Figure 9(g), node 3 is scheduled to stage 1 (p_0), and kept until used by node 4 on PE3, through a “routing PE” denoted by “3r”. This schedule now produces correct results as shown in Figure 9(h).

5.3.4. Backing Up Output Registers. One last precaution to make our Outerloop Pipelining correct is register backup. When EIC is moved and inserted between kernel and epilog, we are actually splitting between them. Certainly there can be many data flows between kernel and epilog, which are typically realized through PE-to-PE interconnections. To preserve the data flows we should back up output registers of PEs, which are the starting points of PE-to-PE interconnections. Backing up output registers is very similar to having shadow registers. Two sets of output registers are maintained and switched between at every transition from kernel to EIC or PIC to kernel.

In addition, EIC and PIC should not destroy other state information, most notably predicate registers, unless it is backed up. Predicate registers, being small, can be easily backed up, or can be reconstructed easily if in-loop conditionals are not used. Storage elements such as PE registers and the local memory need not be backed up, since we avoid destroying live variables through dependency analysis.

Summarizing this section, our scheme of Nested Loop Pipelining (NLP) consists of a set of architectural and compiler techniques. On the architecture side, the CGRA control should include second level looping support (e.g., LC2), the support for sequential execution as illustrated in Algorithm 2, and the support for the instant back-up of output registers. The CGRA compiler has to be extended to generate sequential code, find EIC and PIC, and handle anti- and output dependencies.

Finally, common loop transformations such as loop interchange and loop unrolling are not necessarily competing with our NLP; rather, they can be orthogonal and even complementary. For instance, in a deeply nested loop, if the inner loop’s trip count is small, it may be better simply to fully unroll the innermost loop (and apply single-loop pipelining to the next loop level) than to apply NLP. However, NLP could still be applied to the remaining loop nest. Moreover, since NLP can support only up to two loop

levels, removing the low-trip-count loop beforehand using loop unrolling could actually increase the effectiveness of NLP (compared to applying NLP to the original loop nest).

6. EXPERIMENTS

6.1. Experimental Setup

To evaluate the effectiveness of our nested loop optimization techniques, we use important loops from multimedia benchmarks and scientific computation domain (Applu application from SPEC 2000 [Henning 2000]). Our loops include both perfect and imperfect ones, and their innerloop trip counts are shown in Table IV. The target architecture is a system consisting of ARM9 as the main processor and a CGRA as the only accelerator, connected through an L2 cache and an AMBA2 AHB bus [AMBA2], as illustrated in Figure 3. The CGRA coprocessor has a 4x4 PE array which includes 4 multiplier PEs and 4 load-store PEs (one in each row). Every PE has a small register file (4 entries each) and can perform simple arithmetic operations on operands provided by its 8 neighbors, as illustrated in Figure 1. The 32-entry rotating predicate register file is shared by all the PEs, and can be read by any PE but modified by only 4 PEs (one in each row), in addition to the CGRA control unit.

We generate code for ARM9 processor using the compiler included in the ARM Developer Suite (ADS) [ADS]. CGRA configurations are generated using a version of modulo scheduling [Park et al. 2008], for inner loop body as well as sequential code. We measure the main processor runtime and the CGRA runtime in number of *main processor* cycles, using the ARM9 instruction set simulator included in the ADS and our CGRA simulator. The clock speed of the CGRA is assumed to be half that of the ARM9. The L2 cache, which is based on ARM L220 L2 cache controller [L220], has 4-cycle latency for a noncacheable read or write transaction. The AHB bus adds 1 bus cycle to it, with the bus running at one-third the frequency of the ARM processor. The CGRA is assumed to respond immediately to a “write” transaction (e.g., CGRA parameter setting), but take two CGRA cycles when data must be returned from the PE array (e.g., reading results). To summarize, the communication latency for scalar data between the ARM and the CGRA is about 11 cycles ($4 + 1 \cdot 3 + 2 \cdot 2$) for a read transaction, and 7 cycles ($4 + 1 \cdot 3 + 0$) for a write transaction. Later we vary the communication latency to see the sensitivity of our results on the communication latency.

In all our experiments we assume that necessary CGRA configurations are already in the configuration cache as they can be often prefetched in advance to hide the latency. We also assume that the input arrays are in the CGRA local memory prior to loop execution, or double buffering is used [Singh et al. 2000] for larger arrays to hide data transfer latency.

6.2. Performance Improvement

We compare the execution times of entire nested loops for the three cases outlined in Section 5. In *Baseline* case, only the innermost loop is executed on CGRA while the rest of the loop nest including setup for the innerloop pipeline is done on the main processor. *HWOL* (Hardware Outerloop) case moves the outerloop control and sequential code execution to CGRA, without much effort to compact sequential configurations, and *OLP* (Outerloop Pipelining) case additionally applies sequential code rescheduling and overlaps outerloop iterations through epilog-prolog merging. In addition, we define *SW only* (Software Only) case as a full-software implementation on ARM9. Note that SW only is only for a reference, and our main comparison is against Baseline. For loops with more than two levels (e.g., MatMult) we apply our techniques only to the two innermost levels.

Fig. 10. Performance improvement by nested loop pipelining (Prefix “A_” indicates Applu loop).

Figure 10 shows the runtime results. The runtime of Baseline consists of two parts: (i) CGRA Innerloop is the CGRA execution time for the inner loop including prolog, kernel, and epilog; and (ii) Proc Code + Proc Comm is the software execution time on the main processor, where Proc Comm represents the increased delay due to communication between main processor and CGRA coprocessor while Proc Code is the processor execution time without such communication delay. In comparison, Software Only has only two components: inner- and outer-loop execution times.

Comparing Software Only and Baseline, we observe that mapping only the innermost loop of a loop nest to CGRA can require quite a number of processor cycles for pipeline setup and outer loop execution, though the innerloop’s execution time itself is reduced considerably, more than half on average, compared to full-software implementation. In other words, the innerloop’s execution time is reduced by CGRA mapping, at the cost of increasing the outerloop’s execution time. The overhead of pipeline setup and outer loop execution seems very large, as a combined result of increased overhead (SW Outerloop $\rightarrow$ Proc Code + Proc Comm) and reduced innerloop execution time (SW Innerloop $\rightarrow$ CGRA Innerloop).

Thus it is not surprising to see the largest performance improvement in Hardware Outerloop, or mapping the entire loop nest onto CGRA. This is mainly because the software components have been drastically reduced. By mapping the entire loop nest, we can not only reduce communication delay (such as latency of data transfer instructions), but also eliminate the need for the data transfer instructions. On the CGRA side, Hardware Outerloop runtimes have the same CGRA Innerloop values as in Baseline, but have a new component, CGRA Sequential, which is due to the extra work on CGRA needed to execute outerloop code (array address initialization, Pre- and Post-innerloop statements, etc.). Overall the total runtime is reduced by about 31.6% on average compared to Baseline.

We observe further reduction in runtimes in Outerloop Pipelining, of about 29.9% on average compared to Hardware Outerloop, or about 52.0% on average compared to Baseline. This is due to the overlapping of outerloop iterations, especially epilog-prolog merging, as well as more aggressive rescheduling in generating the sequential configurations.⁸ What may be less obvious is the reduction in software overhead, of setting up the nested-loop pipelines, compared to Hardware Outerloop. This is due to the fact that in Outerloop Pipelining there are far less parameters needed to set up

⁸In Hardware Outerloop, sequential configurations are generated in the order of appearance in the source code, without much rescheduling.

Table IV. Speed-up over ARM9 Only

Loop	Type	N	Baseline	HWOL	OLP
Laplace	P	10	0.90	1.58	2.07
MatMult	P	10	1.27	1.75	2.35
Jpeg_decode	P	8	0.63	0.77	0.99
FIR	I	10	1.26	2.28	3.50
A_setiv_	I	5	1.70	3.27	4.87
A_l2norm_	P	5	1.04	1.33	1.80
A_buts_body	I	5	1.23	1.55	2.08
A_buts_block_inv	I	1~5	0.84	1.03	2.07
A_buts_back_sub	I	1~5	0.47	0.80	1.11

Type: loop nest type (P : Perfect, I : Imperfect).
N: Innerloop's trip count.

Fig. 11. Effect of communication latency on performance. Runtime reduction ratios by our methods over Baseline are shown next to the 3rd and 4th bars in each configuration.

a pipeline than in Hardware Outerloop, thanks to a more compact execution flow in nested-loop pipelining (e.g., four sequential configurations in Hardware Outerloop can be restructured into two in Outerloop Pipelining).

Table IV summarizes the speed-up of different schemes over software-only implementation. Again, for nested loops, innermost loop pipelining (Column 4) doesn't always generate higher performance than Software Only, though innermost loop's execution time itself gets significantly reduced. This means that some nested loops, such as Jpeg_decode and A_buts_back_sub, may better be mapped to the main processor in the conventional approach. In contrast, our nested-loop pipelining can consistently achieve higher performance than Software Only, thus increasing the applicability of CGRAs.

6.3. Effect of Communication Latency

To see the effect of communication latency on the performance of our proposed scheme, we repeat the same experiments for two other sets of communication parameters (see Figure 11). The *Short Latency* is the configuration where the L2 cache latency is removed (originally 4 processor cycles) and the bus is assumed to take one cycle only (i.e., 3 processor cycles), which represents the lowest possible communication latency. In this configuration a single-word read or write transaction takes 7 or 3 processor cycles, respectively. The *Long Latency* is the configuration where the L2 cache latency is the same as in the original configuration but the bus takes two cycles (i.e., 6 processor

Table V. Effect of Innerloop Trip Count

Loop	$N = 8$		$N = 32$		$N = 128$	
	SW	RR	SW	RR	SW	RR
Laplace	59%	60%	32%	35%	11%	13%
MatMult	58%	46%	31%	31%	11%	11%

SW: Software portion in Baseline's runtime

RR: Runtime Reduction over Baseline

cycles), increasing the latency of a single-word read or write transaction to 14 or 10 processor cycles, respectively.

The runtime results for the three configurations are summarized in Figure 11, averaged for all benchmarks. Runtime is normalized to the Software Only case, which doesn't involve CGRA and therefore should be the same in all configurations. As expected, compared to the systems with shorter communication latency, systems with longer latency spend more time in communication (see Proc.Comm in the graph), increasing the overall execution time consequently. But even in the Short Latency case, the runtime reduction by our methods over Baseline is significant, ranging from 26% to 48%, and this reduction ratio increases as the communication latency increases. One of the reasons why the runtime reduction is significant even in the Short Latency case is that the conventional scheme (Baseline) has significant software execution overhead, which is nearly eliminated by our scheme (see Proc.Code in the graph). This demonstrates that our scheme can effectively overcome a certain limitation due to communication inherent in the CPU-CGRA hybrid architecture.

6.4. Trip Count Effect

For some benchmarks the innerloop's trip count is parameterized, which we use to see the effect of the trip count on performance. Table V lists, for different values of parameter N , which is both inner- and outer-loops' trip count, (i) software runtime in Baseline, and (ii) the runtime reduction by our Outerloop Pipelining over Baseline. As expected, the performance improvement of pipelining the entire nested loop diminishes as the innerloop's trip count grows. Put in another way, single-loop pipelining becomes almost as good as nested-loop pipelining if the innermost loop has many iterations. Also the performance improvement by our method closely follows the software runtime of Baseline for a wide range of N , which demonstrates that our method can effectively complement a certain performance overhead of the hybrid CPU-CGRA architecture.

6.5. Energy Reduction

Our proposed methods (Hardware Outerloop and Outerloop Pipelining) require some hardware modification. Though it is small, the extra hardware may offset the energy advantage of our compiler-architecture cooperative scheme to some degree. To quantitatively assess the energy overhead of our methods, we compare the energy consumption of all four cases for each benchmark. For this comparison we use the CGRA power parameters summarized in Table VI, which are from the RSPA architecture [Kim et al. 2005], synthesized for a 180 nm technology [DongbuAnam Semiconductor], adjusted to our PE array size. In the table the memory bank access is for the CGRA local memory, and its power is estimated using Cacti 5.1 [Thoziyoor et al. 2008] and scaled for 180 nm technology.

Our scheme requires additional hardware logic, which mainly consists of the outer-loop trip counter (LC2), counters for sequential code execution (EIC, PIC), and a few extra states in the control FSM to sequence the outer loop. Therefore we assume the overhead, in terms of power as well as area, to be no more than that of one PE, which includes an ALU, muxes, and a small set of registers. But to be more conservative we

Table VI. CGRA Power Parameters

Module	Power Dissipation
Active PE (ALU)	2.543mW
Active PE (Multiplication)	3.200mW
Active PE (Division)	3.465mW
Active PE (Routing only)	0.847mW
Idle PE	0.254mW
The rest part of PE array	25.988mW
Memory bank access	270.03mW
Configuration cache access	34.837mW

Fig. 12. Energy consumption of different schemes.

use twice the power of a PE (doing an ALU operation) as the power overhead due to our scheme, which is added to the rest part power value for Hardware Outerloop and Outerloop Pipelining only.

The ARM processor is assumed to be in either active mode (dissipating 240 mW) or low-power mode (60 mW). The low-power mode may require up to hundreds of cycles on each entry and exit [Lee and Shrivastava 2008], and is therefore utilized only during CGRA execution. We assume for simplicity that ARM dissipates during mode transitions as much power as in the low-power mode. While this evaluation considers dynamic power only, our proposed scheme has advantage in terms of leakage power as well, because it can reduce runtime quite significantly. This evaluation uses the original configuration as used in Section 6.2, and does not include the energy consumption due to, if any, DMA operations between the main memory hierarchy and the CGRA local memory.

Figure 12 shows the energy consumption normalized to that of Baseline. In the Software Only case, ARM is active all the time. The active ARM energy is reduced greatly going from Software Only to Baseline and to Hardware Outerloop and Outerloop Pipelining, proportionally to the processor execution time (Proc.Code plus Proc.Comm in Figure 10). At the same time, however, the low-power ARM energy starts to appear in Baseline, and is increased in Hardware Outerloop and Outerloop Pipelining. But since Baseline has many short low-power intervals whereas Hardware Outerloop and Outerloop Pipelining have a few, very long ones, this increase in the low-power mode energy can be actually much lower if we take into account the energy overhead of mode transitions. Moreover, the low-power ARM energy in Outerloop Pipelining is reduced from Hardware Outerloop, as the CGRA execution time is reduced.

The CGRA energy consumed during sequential computation and during inner loop execution is proportional to the corresponding execution times. A closer examination reveals that the sequential computation, which exists only in the proposed methods,

Table VII. Number of CGRA Configurations

	# Sequential Conf.		# Kernel Conf.
	HWOL	OLP	
Laplace	5	4	5
MatMult	3	2	2
Jpeg_decode	3	2	2
FIR	3	1	2
A_setiv_	9	7	6
A_l2norm_	4	3	2
A_buts_body	8	6	4
A_buts_block_inv	7	3	2
A_buts_back_sub	6	4	2

consumes much less energy than the inner-loop computation despite their similar runtimes. This is because sequential computation uses only one PE at a time and rarely uses memory or multiplication/division operations. Between Baseline vs. Hardware Outerloop, the CGRA inner-loop energy consumption does not change significantly, even after taking into account the extra hardware overhead of our scheme. This is because CGRA energy consumption is dominated by numerous computation operations as well as data and configuration memory accesses, and suggests that our schemes can increase the performance and energy-efficiency of CGRAs at a very marginal cost.

6.6. Configuration Size

Executing outer loop codes on CGRA itself, as proposed by our scheme, has one negative side effect that it requires more configurations. Since configuration caches often have a very limited capacity, with the number of configuration cache entries in recent CGRAs ranging from 16 to 32 [Singh et al. 2000; Kim et al. 2006; Miyamori and Olukotun 1998], it is important to reduce the number of configurations required of a loop. This can be achieved by our epilog-prolog merging (OLP case).

Table VII compares the number of configurations for the sequential computation in HWOL case and OLP case. A comparison between Columns 2 and 3 reveals that our outerloop pipelining can reduce the number of sequential configurations significantly, compared to a straightforward mapping. Column 4 lists the number of innerloop pipeline configurations, or the initiation interval of the innerloop. Therefore, the sum of Column 3 and Column 4 is the total number of configurations, which is in each application no more than 13, indicating that our nested-loop pipelining can consistently generate CGRA mappings that require only a modest number of configurations.

7. CONCLUSION

In this paper we presented nested-loop pipelining for CGRAs, which are well-suited for compute-intensive applications. Unlike single-loop pipelining, nested-loop pipelining must deal with sequential code execution on a CGRA due to the existence of preheader and other initialization code, even in perfect loops. By mapping an entire loop nest onto CGRA, which is extended to support multiple loop levels and intervening sequential executions, we can achieve very high performance improvement of up to 2.87 times compared to mapping innermost loops only, which is mainly due to reduced communication needs between main processor and CGRA coprocessor. Moreover, additional performance improvement can be achieved by overlapping outerloop iterations through aggressive rescheduling of sequential configurations, enabled by our dependency analysis. The configuration size requirement of nested-loop pipelining is only modest, being at most 13 configurations for the loop nests used in our experiments.

Another significance of our nested-loop pipelining is that it enables further research on CGRA mapping such as how to maximize 2D array reuse in nested loops such as

matrix multiplication, which can be more important for architectures with limited local memory size and bandwidth. We also plan to extend our work to support loops with multiple exits.

REFERENCES

- ADS. ARM Developer Suite. ARM Limited. <http://www.arm.com/>.
- AMBA2. Advanced Microcontroller Bus Architecture 2. ARM Limited. <http://www.arm.com/>.
- BONDALAPATI, K. 2001. Parallelizing dsp nested loops on reconfigurable architectures using data context switching. In *Proceedings of the Design Automation Conference*.
- CALLAHAN, T. J. AND WAWRZYNEK, J. 2000. Adapting software pipelining for reconfigurable computing. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'00)*. ACM, New York, 57–64.
- DONGBU ANAM SEMICONDUCTOR. <http://www.dsemi.com>.
- DUTTA, H., KISSLER, D., HANNIG, F., KUPRIYANOV, A., TEICH, J., AND POTTIER, B. 2009. A holistic approach for tightly coupled reconfigurable parallel processors. *Microprocess. Microsyst.* 33, 53–62.
- FELLAHI, M. AND COHEN, A. 2009. Software pipelining in nested loops with prolog-epilog merging. In *High Performance Embedded Architectures and Compilers*, A. Sezenc, J. Emer, M. OBoyle, M. Martonosi, and T. Ungerer, Eds., Lecture Notes in Computer Science Series, vol. 5409, Springer Berlin, 80–94. 10.1007/978-3-540-92990-1_8.
- H.264. H.264/AVC JM reference software v15.0. <http://iphome.hhi.de/suehring/tml/>.
- HANNIG, F., DUTTA, H., AND TEICH, J. 2004. Mapping of regular nested loop programs to coarse-grained reconfigurable arrays - constraints and methodology. In *Proceedings of the 18th International Symposium on Parallel and Distributed Processing*. 148.
- 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, Los Alamitos, CA, 642–649.
- HENNING, J. 2000. SPEC CPU2000: measuring cpu performance in the new millennium. *Computer* 33, 7, 28–35.
- HUCK, J., MORRIS, D., ROSS, J., KNIES, A., MULDER, H., AND ZAHIR, R. 2000. Introducing the IA-64 architecture. *IEEE Micro* 20, 12–23.
- KIM, Y., KIEMB, M., PARK, C., JUNG, J., AND CHOI, K. 2005. Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'05)*. IEEE Computer Society, Los Alamitos, CA, 12–17.
- KIM, Y., PARK, I., CHOI, K., AND PAEK, Y. 2006. Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architecture. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'06)*. ACM, New York, 310–315.
- L220. ARM L220 cache controller technical reference manual. ARM Limited. <http://infocenter.arm.com/>.
- LEE, J. AND SHRIVASTAVA, A. 2008. Static analysis of processor stall cycle aggregation. In *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'08)*. ACM, 25–30.
- MEI, B., VERNALDE, S., VERKEST, D., DE MAN, H., AND LAUWEREINS, R. 2002. DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*. 166–173.
- MEI, B., VERNALDE, S., VERKEST, D., DE MAN, H., AND LAUWEREINS, R. 2003. 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)*. IEEE Computer Society, Los Alamitos, CA.
- MIYAMORI, T. AND OLUKOTUN, K. 1998. Remarc: Reconfigurable multimedia array coprocessor. *IEICE Trans. Info. Syst.* E82-D, 389–397.
- MUCHNICK, S. S. 1997. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers.
- MUTHUKUMAR, K. AND DOSHI, G. 2001. Software pipelining of nested loops. In *Proceedings of the 10th International Conference on Compiler Construction (CC'01)*. Springer-Verlag, Berlin, 165–181.
- PARK, H., FAN, K., MAHLKE, S. A., OH, T., KIM, H., AND KIM, H.-S. 2008. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. ACM, New York, 166–176.
- PARK, H., PARK, Y., AND MAHLKE, S. 2009a. Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*. 370–380.

- PARK, Y., PARK, H., AND MAHLKE, S. 2009b. Cgra express: accelerating execution using dynamic operation fusion. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'09)*. ACM, New York, 271–280.
- PATTERSON, D. AND HENNESSY, J. 2008. *Computer Organization and Design: The Hardware/Software Interface* 4 Ed., Morgan Kaufmann.
- PETKOV, D., HARR, R., AND AMARASINGHE, S. 2002. Efficient pipelining of nested loops: unroll-and-squash. In *Proceedings of the International Symposium on Parallel and Distributed Processing Symposium (IPDPS'02)*. Abstracts and CD-ROM.
- RAU, B. R. 1994. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO 27)*. ACM, New York, NY, 63–74.
- RONG, H., TANG, Z., GOVINDARAJAN, R., DOUILLET, A., AND GAO, G. 2004. Single-dimension software pipelining for multi-dimensional loops. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. 163–174.
- SINGH, H., LEE, M.-H., LU, G., BAGHERZADEH, N., KURDAHI, F. J., AND FILHO, E. M. C. 2000. Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Comput.* 49, 465–481.
- THOZIYOOR, S., MURALIMANOHAR, N., AHN, J., AND JOUPPI, N. 2008. Cacti 5.1. Tech. rep.
- WANG, J. AND GAO, G. R. 1996. Pipelining-dovetailing: A transformation to enhance software pipelining for nested loops. In *Proceedings of the 6th International Conference on Compiler Construction*. Springer-Verlag, Berlin, 1–17.

Received July 2011; revised October 2011; accepted November 2011