Exploiting Thread-Level Parallelism in Lockstep Execution by Partially Duplicating a Single Pipeline

Jaegeun Oh, Seok Joong Hwang, Huong Giang Nguyen, Areum Kim,
Seon Wook Kim, Chulwoo Kim, and Jong-Kook Kim

In most parallel loops of embedded applications, every iteration executes the exact same sequence of instructions while manipulating different data. This fact motivates a new compiler-hardware orchestrated execution framework in which all parallel threads share one fetch unit and one decode unit but have their own execution, memory, and write-back units. This resource sharing enables parallel threads to execute in lockstep with minimal hardware extension and compiler support. Our proposed architecture, called multithreaded lockstep execution processor (MLEP), is a compromise between the single-instruction multiple-data (SIMD) and symmetric multithreading/chip multiprocessor (SMT/CMP) solutions. The proposed approach is more favorable than a typical SIMD execution in terms of degree of parallelism, range of applicability, and code generation, and can save more power and chip area than the SMT/CMP approach without significant performance degradation. For the architecture verification, we extend a commercial 32-bit embedded core AE32000C and synthesize it on Xilinx FPGA. Compared to the original architecture, our approach is 13.5% faster with a 2-way MLEP and 33.7% faster with a 4-way MLEP in EEMBC benchmarks which are automatically parallelized by the Intel compiler.

Keywords: ILP, TLP, SMT, CMP, MLEP.

Manuscript received Dec. 12, 2007; revised June 19, 2008; accepted June 30, 2008.

This research was supported by grant No. R01-2005-000-10124-0 from the Basic Research Program of the Korea Science and Engineering Foundation, and Nano IP/SoC Promotion Group of R&BD Program in 2008.

Jaegeun Oh (phone: 82 2 3290 3892, email: worms97@korea.ac.kr), Seok Joong Hwang (email: nzthing@korea.ac.kr), Huong Giang Nguyen (email: redriver@korea.ac.kr), Areum Kim (email: naareumi@korea.ac.kr), Seon Wook Kim (phone: + 82 2 3290 3251, email: seon@korea.ac.kr), Chulwoo Kim (email: ckim@korea.ac.kr), and Jong-Kook Kim (email: jongkook@korea.ac.kr) are with the School of Electrical Engineering, Korea University, Seoul, Rep. of Korea

I. Introduction

Intrinsically embedded applications have a high degree of parallelism on several levels, such as data, instruction, and thread levels [1], [2]. However, it is difficult to exploit that parallelism because real embedded platforms are composed of very simple cores due to manufacturing cost, chip area, power consumption, and thermal dissipation. Therefore, it is very costly and maybe impractical to use the state-of-the-art high performance processors like superscalars [3], simultaneous multithreading [4], and so on, in embedded systems. One promising approach is to use several simple processors on one chip such as chip multiprocessors (CMPs). Recently, many industry vendors have introduced CMPs targeted for embedded applications [5], [6]. The CMP consumes less energy than superscalar and symmetric multithreading (SMT) architectures in applications to exploit high thread-level parallelism, but still consumes much power and resources. For example, the power consumption and resource complexity in the two processors of the ARM11^{TM} MPCore are 2.25 and 2.17 times higher than those of a single core.

Another attractive approach to achieve high performance in embedded codes is to employ single-instruction multiple-data (SIMD) execution. We experimentally applied the Intel compiler 9.1.042 to automatically simdize several of the EDN Embedded Microprocessor Benchmarking Consortium (EEMBC) benchmarks [7] selected from [8]. Figure 1 shows the percentage of the total execution time spent in simdized and parallelized regions automatically identified by the compiler, and it was very low. For measurement, we used the number of executed cycles on our synthesized experimental processor, AE32000C, on FPGA [9]. The compiler cannot exploit data level parallelism (DLP) in many highly parallel loops due to

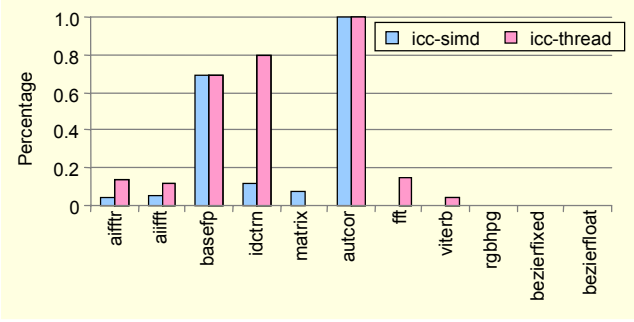


Fig. 1. Percentage of the total execution time spent in simdized and parallelized loops automatically identified by the Intel compiler.

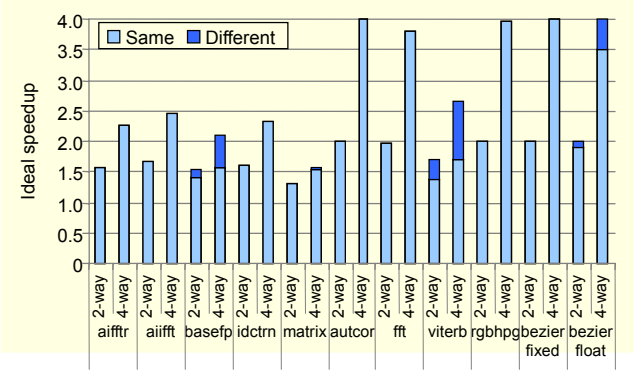


Fig. 2. Upperbound of speedup on our architecture with 2 and 4 threads in hand-parallelized codes.

their enclosing pointers, mixed data types, function calls, branches, dependences, and so on, especially in rgbhpg, bezierfixed, and bezierfloat [10]. It is well known that it is not possible to simply simdize codes without aggressively restructuring and rewriting applications [11], [12]. The Intel compiler also fails to identify parallel loops due to unknown loop trip counts, unidentifiable private variables, and so on. In our test result, however, it exploits higher parallelism in loop parallelization (icc-thread in Fig. 1) than in simdization (icc-simd). The compiler can automatically identify loop-level parallelism (LLP) in almost all simdizable and non-simdizable loops. The DLP is normally expressed in forms of loops in embedded codes. It is easier for the compiler and programmer to detect LLP than DLP. By eliminating the limited applicability of SIMD instruction set architecture (ISA) and parallelizing the outermost loops rather than the innermost ones, thread-based parallel execution from LLP can express more parallelism than SIMD/vector execution in general.

In this paper, we propose a new processor architecture, called multithreaded lockstep execution processor (MLEP), which overcomes the resource cost of CMPs and the limited applicability of simdization of SIMD machines. Our architecture exploits thread-level parallelism (TLP) as in SMT/CMP from LLP, but it executes parallel threads in lockstep as in SIMD by translating TLP into statically scheduled instruction level parallelism (ILP) by a compiler. On CMPs and SMTs, each thread executes its own code sections independently with appropriate synchronizations. In most parallel loops of embedded applications, however, every iteration executes the exact same sequence of instructions while processing different data. If a group of threads executes the identical code sequence, transferring the code to processors and then fetching and decoding them separately is not an optimal method. Instruction fetch and decode units can be shared among threads. Each thread only needs private execution, memory, and write-back units. Sharing allows parallel threads to execute in lockstep, as in SIMD execution. Our approach has advantages over SIMD execution in terms of applicable range and code generation because it can use normal instruction sets without restriction. Also, by using one instruction fetch unit and one instruction decode unit for parallel executions, our approach use less power and chip area than a chip multiprocessor without significant loss of performance in embedded applications.

Our proposal is based on the following observations. Most embedded applications can be highly parallelized to exploit TLP. Parallel loops consume the most execution time, and all threads execute the exact same code sequence in most cases. We marked the starting and ending points of parallel loops identified in EEMBC benchmarks [8], and measured the number of total execution cycles to calculate the potential speedup [9] on our experimental processor. Figure 2 shows an upperbound speedup in two categories by applying Amdahl's Law with 2 and 4 threads. The category "same" indicates the ideal speedup achieved by parallel codes which do not include any if-else/switch clause; therefore, it is guaranteed that all threads execute the same code sequence. That is, code sequences are syntactically identical. The category "different" indicates speedup when it is assumed that two threads execute the same code sequence even if there are if-else/switch clauses inside parallel loops. The achieved speedup is very high. All benchmarks have an ideal speedup of 1.76 in the 2-way MLEP and 3.01 in the 4-way MLEP on average, and five of them achieve ideal speedup.

Figure 3 shows the distribution of dynamic instructions in parallel and serial regions. The ratio of branch instructions in parallel regions is less than 1%, which implies that the possibility of control-flow conflict between threads is very low. In other words, there are very few conditional clauses, such as if-else/switch clauses. They are in only in four of the benchmarks in Fig. 2. This implies that parallel threads execute the same code sequence most of the time. Memory instructions in parallel regions are about 20% of the total instructions on

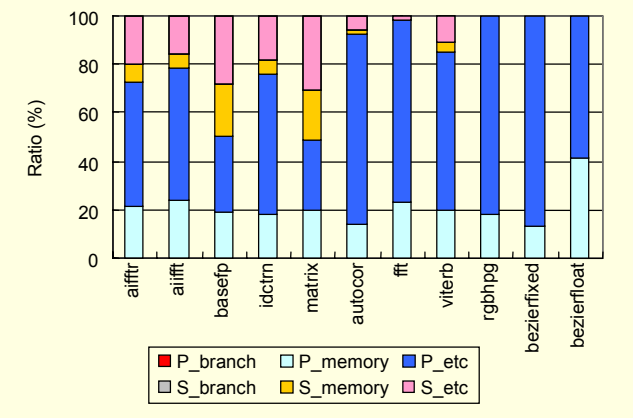


Fig. 3. Distribution of dynamic instructions in parallel (P_*) and serial (S_*) code sections in Fig. 2.

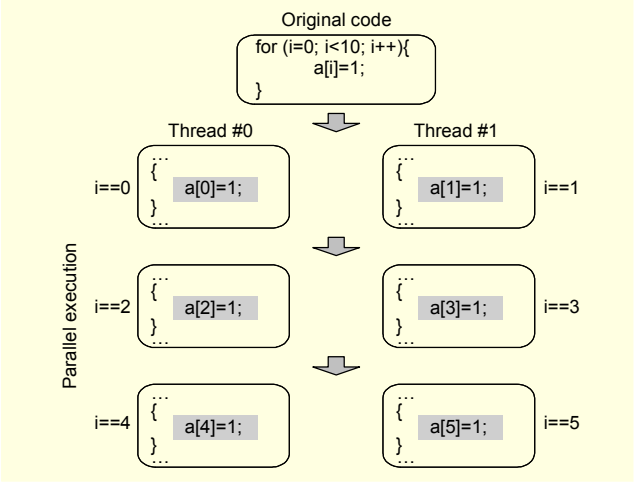


Fig. 4. Execution of the same control-flow in a parallel loop. The shaded boxes represent the executed code sections in the loop body.

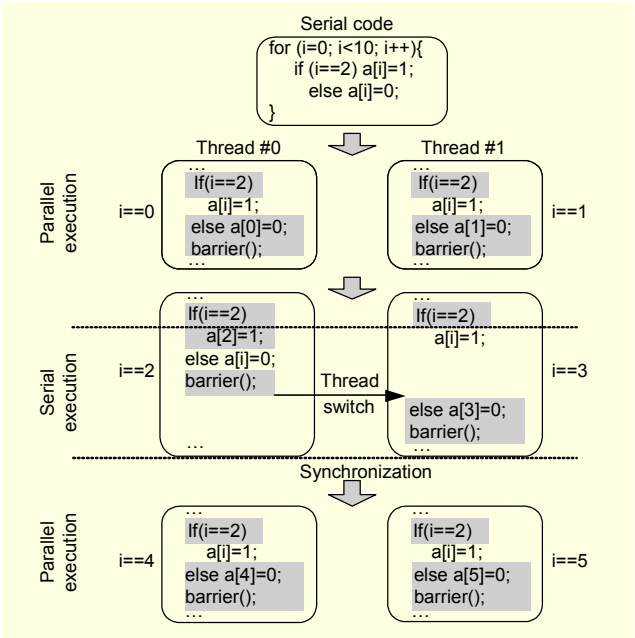


Fig. 5. Execution of a different control-flow in a parallel loop. The shaded boxes represent the executed code sections in the loop body.

average. Therefore, a data cache is needed which can process multiple memory instructions from threads simultaneously to achieve good performance.

Our experiment shows that in comparison with the original architecture, our proposed architecture achieved speeds 1.14 and 1.34 times faster on average and up to 1.98 and 3.86 times faster in automatically parallelized EEMBC embedded benchmarks with increases of 62.2% and 182.9% in complexity and 41.9% and 124.9% in power consumption with 2-way MLEP and 4-way MLEP, respectively. Speed improvements of 1.44 and 1.83 were also achieved in hand-parallelized codes on 2-way MLEP and 4-way MLEP, respectively.

The remainder of this paper is organized as follows. In section II, we introduce the overall organization of our architecture, and in the next section, we present its implementation. Section IV analyzes its performance. Related works are discussed in section V, and the last section summarizes our research.

II. Architecture Overview

We explain the execution behavior of our architecture, MLEP, with examples of 2-thread execution. If two threads execute the same instruction sequence as shown in Fig. 4, one instruction fetch unit and one decode unit are sufficient for multithreading execution (see section III.3.A for details). When two threads execute different control-flows, only one thread becomes active at a time, and in a predefined scheduling order, as shown in Fig. 5. When the divergence ends, two threads are synchronized for their following simultaneous execution. We need a mechanism that threads know where they are synchronized. We introduced a barrier instruction, inserted by our compiler, to mark a synchronization point, where threads are guaranteed to join. This approach does not incur large overhead, since the number of executed branch instructions is very small as shown in Fig 3. This case also covers the odd number of loop iterations (see section III.3.B for details) In general, there is high spatial data locality of memory references in two sequences of loop iterations. Therefore, in order to use one shared data cache and to exploit high spatial locality, we use cyclic scheduling. In Fig. 4, a constant 1 is stored in both a[i] and a[i+1] at the same time; therefore, we may store two words at one time with address a[i] instead of accessing the data cache twice. We can take this opportunity to achieve better speedup only with small modification of an interface between a data cache and a core, and its details are described in section III.4.

In order to verify the above working principle, we

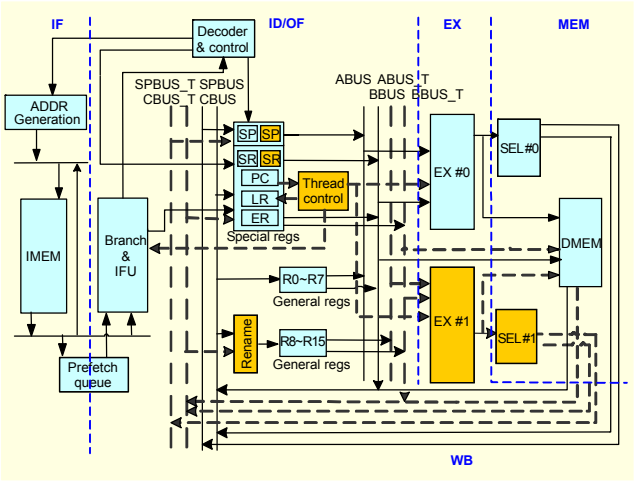


Fig. 6. Hardware architecture overview for 2-way MLEP. The cross-lined boxes and dashed lines were added for our purpose.

implemented the idea by modifying the AE32000C core which is a standard single 5-stage pipeline microprocessor architecture [9]. The overview of our architecture for 2-way extension is shown in Fig. 6, and our modification is marked as cross-lined boxes and dashed lines. Our prototyped processor supports 2 to 4 threads. However, since our solution can handle control-flow conflicts for any number of threads, it is completely scalable. Also, the achieved speed improvement is scalable with the number of cores. As shown in Fig. 6, the fetch unit was kept the same as that in the original architecture. The decode unit was changed a little to support thread control instructions. The memory and write-back units were modified to support multiple memory accesses at the same time. There are two approaches to developing register files. With one approach original register files are duplicated, and with the other, they are not. This register file duplication problem is explained in detail in section III.2. Also, some special registers and buses were added in our architecture.

III. Architecture Implementation

1. Code Generation

We used a typical code generation scheme for a shared-memory system. OpenMP directives are used to identify parallel loops [13], and the Omni compiler is used to translate OpenMP codes into subroutine-based forms in order to provide stack space to each thread to handle thread private variables [14]. The Omni OpenMP runtime library was modified in order to use our thread libraries instead of PThread. The translated codes are compiled by the gcc-based EISC compiler and linked with our libraries to build executable codes [15].

2. Core Extension for Multithreading

The minimum resources needed for one independent thread execution are an execution unit, a private register file and a separate stack-space. We provided each thread with a private execution unit and a private stack space. However, with the register file we provided two options: a basic approach and an extended approach. In the basic approach, to minimize resource usage and power consumption we do not duplicate the original register file (16 general registers, R0-R15 in AE32000C). The original general register file is divided into several identical banks for parallel execution. In serial execution, the processor uses all general registers, but in parallel, each thread uses a half for two threads and a quarter of all general registers for four threads. We rely on our compiler to generate codes using only a half or a quarter of general registers (R0-R7 or R0-R3) for parallel sections. A register rename unit was added in front of the child threads to rename instruction register operands from R0-R7 to R8-R15 for two threads, and from R0-R3 to R4-R7, R8-R11 and R12-R15 for four threads respectively. Of course, this approach introduces overhead due to register spills, appeared in section IV.2. However, the spill problem can be alleviated by a rich set of register files. In the extended approach, all general registers are duplicated. Some special registers, such as a program counter and an instruction register, are shared. Other special registers, such as a status register and a stack pointer, were added and made privately for each thread.

In the decoding stage, a thread control module was also added to control parallel execution. It enables or disables execution units, manages the program counter register, and handles a branch control unit to detect and manage control-flow conflicts. The interface between a core and a data cache needs to be modified to support simultaneous memory accesses from multiple threads. As a basic implementation, we added a memory controller unit between the core and the data cache to perform the task of sequentially scheduling the multiple memory accesses and sequentially transferring data to the data cache. This makes the memory accessing time longer in parallel execution, which is an obstacle to achieving faster speed for memory instructions of our architecture. The simplest solution is to use a multiple-port data cache; however, in section III.4 we will further optimize this issue by widening the interface between the core and the data cache to exploit spatial locality between threads.

3. Hardware and Compilation Interaction

A. Identical Control-Flow Execution in Parallel Loops

Depending on a thread identification number (ID), each

parallel thread knows which data it is responsible for. This ID is provided through the thread_id function of our library. The thread ID is stored in a special register inside execution units. At the starting point of parallel sections, thread 0 functions as the main thread to initialize the parallel execution environment using the fork function. This function performs the task of copying a thread function's parameters into the threads' private stack space and invokes a processor to start to run in parallel by waking child threads that are dedicated to the added execution units. The parallel execution of our new architecture operates as follows. The fetch and decode units still fetch and decode only one instruction at a time as in serial execution. All execution units receive instructions from the decode unit and process them using their own data. The problem of multiple threads accessing the memory at the same time in memory stages is resolved by a new data cache memory controller unit. After obtaining the results from memory stages, multiple threads update their own register files at write-back stages in parallel. The task of joining threads is also implemented in the fork function. All child threads are stalled and only thread 0 continues running serially after join execution. This ideal parallel execution is applied to loops which have no if/switch clause and have an even number of iterations. If a loop has an odd number of iterations, we consider the last iteration an if-clause. Its implementation is described in the next sub-section.

B. Different Control-Flow Execution in Parallel Loops

Because of having different data, parallel threads can meet different conditions of branch instructions, such as "if" or "switch-case", and no longer execute the same instruction sequence. Because our architecture uses only one instruction fetch unit and one decode unit for all threads, we need to resolve the conflict. The best solution is to divide threads into two groups: the taken branch and the branch not taken. Then, we let each group execute its code sequentially one-by-one until both groups start to execute the same code sequence again. Figure 5 shows an example of this control-flow conflict. This problem never occurs in conventional multithreaded architectures where parallel threads execute independently, possibly on different processors. When a control-flow conflict occurs, we need to solve the following three problems: first, how to detect when a control-flow conflict occurs; second, how to schedule the execution of threads when there is a time conflict; and third, how to determine when the conflict ends, so that we can execute threads in parallel again.

Solving the first problem is quite simple. A control-flow conflict happens only at conditional branches, and their decision depends on the result of the previous comparison instruction. By comparing the status registers of all threads, we can easily detect whether a control-flow conflict occurs. The third problem is more difficult and very complex to solve by hardware alone. The participation of software makes the problem much easier. Our solution uses a special instruction, called a *barrier* function to mark the end of code sections susceptible to conflict, and a compiler is responsible for producing the correct codes using the barrier function. After determining the start and end of conflict codes, the hardware can solve the second problem by sequentially running each thread group's code sequences until it meets a barrier instruction. We inserted a barrier at the end of each loop iteration to synchronize diverged control-flows as quickly as possible. Of course, threads work correctly even if we insert a barrier outside the loop. However, they can experience severe load imbalance if control-flows diverged in the loop iteration.

The scenario becomes more complex when barriers are nested. Figure 7 shows examples of nesting barriers with four threads, where solid-lines represent the execution paths of active threads and dashed-lines represent jumps (and context switching) forced by barriers. We classify control-flow executions into five types, namely, switch, re-switch, merge, resume, and ignore. We maintain two types of records, namely, bar_record and div_record. Bar_record stores an execution of a barrier instruction to be inserted by a compiler. This record consists of a level, a call depth, the next PC, and the masks of threads to reach the barrier. The level implies the depth of nesting control-flow in codes. Div_record keeps control-flow divergence records and uses a stack structure. Whenever threads diverge, one group of threads continues to execute by some scheduling priorities. The others are deactivated and their information is pushed into the stack to be scheduled later. Div_record consists of a call depth, next PC, and the masks of the deactivated threads.

In Fig. 7(a), assume that a control-flow diverges at a branch IF0 at t_0, and threads 0 and 1 continue to execute in the direction of the branch taken. Div_record records 0 as the call depth, L0 as the execution starting point for the other deactivated threads (threads 2 and 3), and 0011 as the thread mask. When active threads (threads 0 and 1) execute a barrier BAR0, the barrier information is recorded in bar_record. Then threads are switched to inactive threads by popping the top of the div_record stack, and the stack becomes empty. Then, assume that a branch IF1 is executed and thread 2 continues to execute at t_2. When the thread (thread 2) executes BAR1, the level of BAR1 (1 in this case) is compared with a barrier record in bar_record. If the level of BAR1 is larger than the barrier record (it implies that BAR1 is enclosed by another barrier in bar_record), BAR1 is ignored. If we allow a context switching for an inner barrier, a dead-lock may occur. In this case, we conservatively resolve the nesting barriers. If we can record multiple barriers, we may resolve the issue more aggressively.

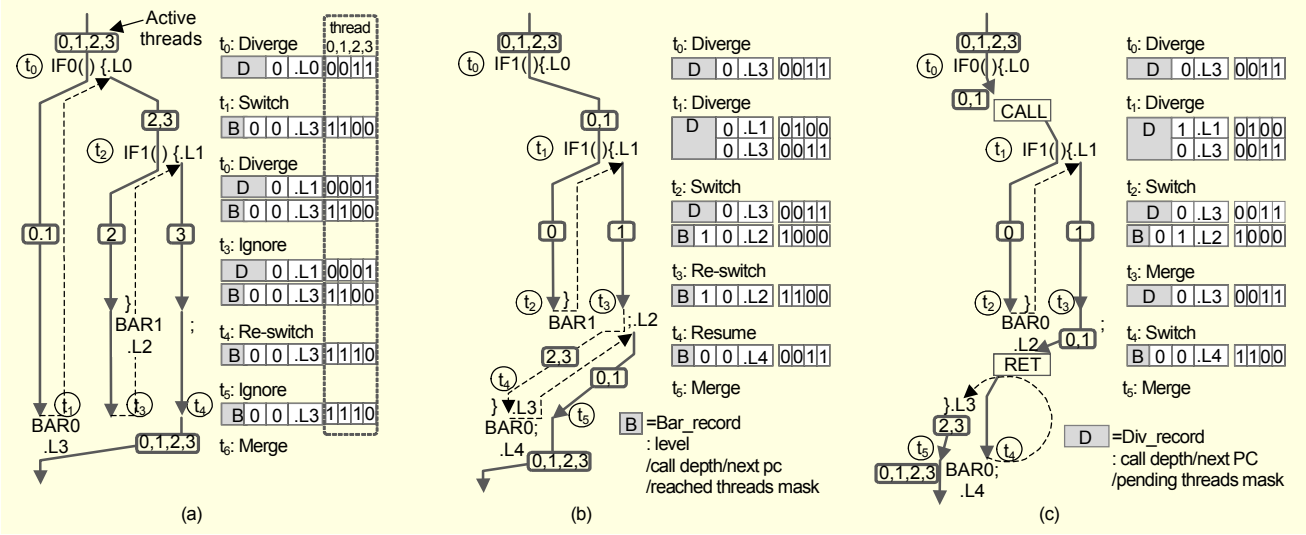


Fig. 7. Examples of control-flow divergence handling.

After ignoring BAR1, thread 2 meets BAR0, which is the same barrier as that in the barrier record, so we take a re-switch action. This action is quite similar to switching except that re-switching just updates the thread mask of the barrier record by an OR operation with the mask of current active threads. Therefore, threads 0, 1, and 2 are recorded in the mask, and thread 3 becomes active. Then, thread 3 meets BAR0 again after ignoring BAR1. All control-flows are merged, and we activate all threads.

Consider Fig. 7(b), where the first branch direction falls inside of the if-clause (IF0), and threads 0 and 1 are active. At IF1, thread 0 continues to execute and finally reaches BAR1. After two consequent control-flow divergences, we take switch and reswitch actions for BAR1. Then, threads 2 and 3 meet BAR0 whose level is lower than the barrier record in bar_record. This indicates that there are un-executed paths caused by the previous re-switch action; therefore, we resume threads deactivated by the previous context switching. Figure 7(c) shows how our algorithm works with function calls. Although two barriers have the same level, they are distinguishable by the call depth of their functions.

The performance, as well as the behavior, of the algorithm depends on which barrier (which control-flow) we execute first. If we schedule threads as shown in Fig. 7(b) (where the first branch direction falls inside an if-clause), we may omit the ignore action and execute multiple threads after BAR1 as in Fig. 7(a). Although we can resolve this imbalance by maintaining barrier records in a stack structure, the proposed algorithm is acceptable. The reason is that the outermost barrier is usually a pair of for-loops with an increasing induction variable. With such for-loops, taken branch directions always fall inside the for-loops because a main thread carries out the last iteration.

This control-flow conflict occurs in both programmers' codes and library functions. To ensure the correctness of parallel execution, we also developed a thread-safe library for our architecture.

4. Widening the Data Cache Interface

A single-port data cache will cause pipeline stalls in every parallel memory access in our architecture. However, threads may access data in the same cacheline in many cases because a cyclic scheduling scheme introduces high spatial locality. To exploit this opportunity, we widened the cache interface bus from 32 bits to 64 bits for 2-way or 128 bits for 4-way extension so that multiple threads can read and write data in only one memory access if their accessing addresses are in the same cacheline.

Figure 8 shows the organization of our data cache and memory controller for 2-way extension. In parallel execution, the controller determines how to access the cache by comparing accessed addresses (ADR0, ADR1) from each thread. If these are in the same cacheline, the memory

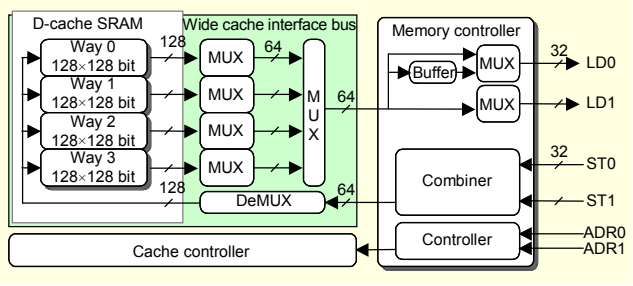


Fig. 8. Widening data cache interface for 2-way extension.

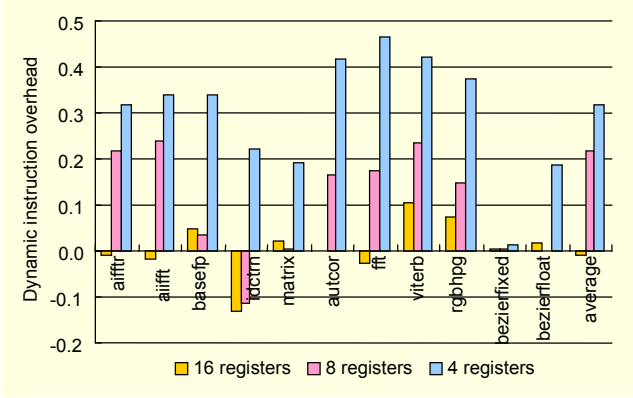


Fig. 9. Dynamic instruction overhead with respect to original codes.

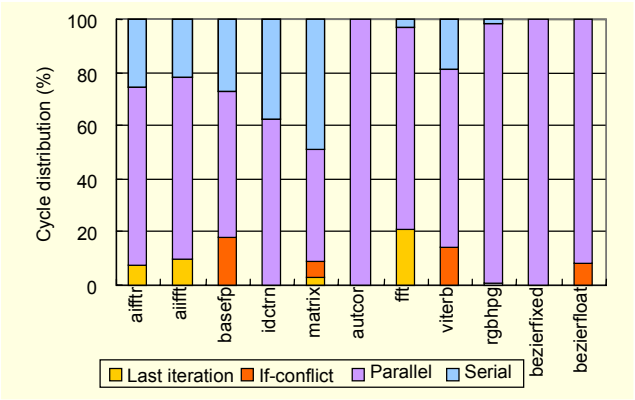


Fig. 10. Cycle distribution of 4-way MLEP with 16 registers when using hand-parallelized codes.

controller multiplexes the incoming 64 bits from the cache to load the whole thread's data (LD0, LD1), or combines their stored data (ST0, ST1) and stores them as single double-sized data in one memory access. Otherwise, two memory accesses are handled sequentially in an interleaved manner, and a buffer is deployed for this purpose. In serial execution, the controller just routes all cache signals to an active thread.

IV. Performance Analysis

1. Experimental Setup

In this section, we evaluate our parallel architecture's performance and compare it to that of the original single architecture and the Intel Duo processor which supports SSE2/SSE3 execution in running EEMBC benchmarks [7]. For performance comparison with SIMD execution, we used the Intel 9.1.042 compiler to simdize the benchmarks and to identify parallel loops. The identified parallel loops are annotated by the OpenMP directives by hand, and are compiled by our compiler infrastructure, that is, the Omni and EISC backend compiler. We parallelized 11 out of 34 benchmarks identified in [8]. We implemented our architecture by extending the commercial EISC AE32000C of ADChips Inc. [9] with the following system parameters: in-order execution, 24 MHz clock speed, I-cache of 8 KB, 4-way 1 cycle hit, 128-bit cache line, D-cache of 8 KB, 4-way 1 cycle read, 2 cycle write, 128-bit cache line, 24 cycles for data, and 42 cycles for instruction memory latency. All experiments were run on an evaluation board using a Virtex 4 XC4VLX100 FPGA of Xilinx.

We compared our performance only with Intel SSE execution since we wanted to use only one parallelizing compiler to generate parallel codes for different architectural variants (SIMD, CMP/SMT, and VLIW) for fairness of performance comparison. The Intel compiler was the only good available candidate since it automatically parallelizes codes for TLP and simdizes them for SIMD execution. Using different compilers for performance comparison would make it difficult to see the architectural advantages due to their different abilities.

Each benchmark is measured in several aspects. The "icc-simd" category means that the codes are automatically simdized by the Intel compiler and executed on the Intel processor. The icc-thread category implies that parallel loops are automatically identified by the Intel compiler but, executed on our platform. Our platform supports *x*-way extension, and each thread uses *y* registers. Similarly, "hand-thread" is the same as "icc-thread" except that the codes are parallelized by hand from [8].

2. Instruction Overhead

There are two kind of instruction overhead in our architecture: parallelization overhead and register spill overhead. The parallelization overhead consists of parallel code overhead and thread management overhead. The parallel code overhead results from code transformation of serial codes into parallel versions (conversion of the OpenMP-directed parallel loops into subroutine-based forms). The thread management overhead is caused by the execution of thread library functions. The parallelization overhead is not avoidable in TLP programming, but the register spill overhead is caused by the lack of available registers in parallel execution due to the minimal extension in the hardware portion of our architecture.

Figure 9 shows the dynamic instruction overhead of hand parallelized codes with respect to serial codes. The instruction overhead of parallelized benchmarks compared to serial ones is around -1.0% for 16 registers, 21.7% for 8 registers, and 32.0% for 4 registers per thread on average. The overhead for 16 registers implies a parallelization overhead, which is very insignificant in most cases except for basefp, viterb, and rgbhpg.

In this case, some benchmarks have negative instruction overhead for 16 registers, since the translated codes from the Omni compiler may allow the gcc compiler to generate better quality of codes.

3. Speedup

Table 1 shows the speedup of Intel SIMD and our proposed architecture, MLEP, with respect to serial execution. In the execution of parallel codes automatically identified by the Intel compiler, MLEP with 4 and 8 registers achieved weaker performance than the Intel SIMD execution due to register spill overhead. But with 16 registers, our approach outperformed Intel SSE2/SSE3 4-way SIMD execution by 4.6% on 2-way MLEP and 22.9% on 4-way MLEP. This better speedup results from higher parallelism as shown in Fig. 1. Only in the matrix benchmark, SIMD execution performs better than MLEP. In hand-parallelized codes, MLEP achieved much higher performance than auto-parallelization. In hand-parallelization, the OpenMP directives are annotated in code by hand are used as input for the compiler infrastructure. The benefit of widening a data cache interface is 9.4% speedup in auto-parallelization and 11.6% speedup in hand-parallelization when using 4-way MLEP with 16 registers.

Because the Intel processor executes multiple instructions per cycle in simdizable code regions, it is difficult to directly compare the performance of MLEP and SIMD execution. Because DLP can be exploited as ILP, the speedup resulting

Table 1. Speedup comparison of our architecture (icc-thread and hand-thread) and SIMD on Duo SSE2/SSE3 (icc-simd) in auto- and hand-parallelized codes.

Benchmarks	Cache	Auto-parallelization by the Intel compiler					Hand-parallelization from [8]			
		icc-simd (4-way)	icc-thread (2-way)		icc-thread (4-way)		Hand-thread (2-way)		Hand-thread (4-way)	
			8 regs	16 regs	4 regs	16 regs	8 regs	16 regs	4 regs	16 regs
aifftr	1 port	0.93	1.11	1.11	1.15	1.16	1.04	1.19	0.97	1.17
	Widening		1.12	1.12	1.17	1.17	1.12	1.30	1.08	1.33
aiifft	1 port	0.98	1.12	1.12	1.14	1.15	0.98	1.26	0.87	1.12
	Widening		1.12	1.12	1.16	1.16	1.04	1.35	0.95	1.27
basefp	1 port	1.05	1.16	1.15	0.79	1.19	1.16	1.16	0.79	1.21
	Widening		1.16	1.15	0.79	1.19	1.16	1.16	0.79	1.21
idctrn	1 port	1.00	0.95	1.00	0.83	1.11	1.49	1.55	1.03	1.82
	Widening		0.98	1.05	0.89	1.23	1.56	1.63	1.09	1.99
matrix	1 port	1.15	1.00	1.00	1.00	1.00	1.12	1.14	0.92	1.18
	Widening		1.00	1.00	1.00	1.00	1.12	1.14	0.92	1.18
autcor	1 port	1.82	1.27	1.75	1.12	2.78	1.27	1.75	1.12	2.78
	Widening		1.39	1.98	1.26	3.86	1.39	1.98	1.26	3.86
fft	1 port	1.00	1.07	1.05	1.00	1.06	0.99	1.40	0.68	1.42
	Widening		1.09	1.07	1.03	1.10	1.09	1.62	0.74	1.74
viterb	1 port	1.00	1.00	1.00	1.00	1.00	0.96	1.22	0.87	1.35
	Widening		1.00	1.00	1.00	1.00	0.96	1.22	0.87	1.35
rgbhpg	1 port	1.00	1.00	1.00	1.00	1.00	1.29	1.43	1.18	1.97
	Widening		1.00	1.00	1.00	1.00	1.29	1.43	1.18	1.97
bezierfix	1 port	1.00	1.00	1.00	1.00	1.00	1.56	1.56	2.15	2.15
	Widening		1.00	1.00	1.00	1.00	1.57	1.57	2.19	2.18
bezierfloat	1 port	1.00	1.00	1.00	1.00	1.00	1.42	1.42	1.16	1.83
	Widening		1.00	1.00	1.00	1.00	1.42	1.42	1.16	1.83
Average	1 port	1.09	1.06	1.11	1.00	1.22	1.21	1.37	1.07	1.64
	Widening		1.08	1.14	1.03	1.34	1.25	1.44	1.12	1.83

from simdization on a multiple-issue processor is usually less than the speedup resulting from simdization on a single-issue processor. However, IPC on the Intel processor was 1.62 with simdization and 1.77 without. This indicates that IPC is very similar in non-simdizable and simdizable regions, and the speedup of SIMD can be compared with 2.5-way (4-way/IPC) of our approach. The performance gap between the ideal (Fig. 2) and the real measurement results from several factors: the parallelization overhead, the register spills shown in Fig. 9, the data cache misses (92.8% hit in 2-way and 90.2% in 4-way), and the control-flow conflict (basefp: 24.7%, matrix: 11.0%, viterb: 11.4%, and 8.5% in bezierfloat of the parallel execution time in 4-way with 16 registers) as shown in Fig. 10.

Our architecture is scalable in terms of performance metrics. Applying Amdahl's Law to the icc-thread 2-way 16-register speedup, we calculate that the parallel region is about 26.9% of the total code. By applying this ratio to Amdahl's Law with 4-way extension, we get the upperbound speedup of 1.25. The higher measured speedup of 1.34 comes from the higher number of data cache hits on the 4-way MLEP. Similarly, the achieved speedup in hand-parallelized codes is 1.83, and the calculated upperbound speedup based on Amdahl's Law is 1.85 (61.1% parallel regions on average) on the 4-way with 16 registers.

Figure 10 shows the cycle distribution on the 4-way MLEP with 16 registers using hand-parallelized codes. In the figure, "serial" means a serial execution, "parallel" means that all threads execute the same control-flow, "last iteration" means that control-flows diverge at the last iterations of parallel loops where there are not enough iterations to be fetched for all threads, and "if-conflict" indicates *real* control-flow conflicts in parallel regions. Only a few benchmarks include real control-flow conflicts, and this strongly supports our motivation. The control-flow conflicts occur in floating-point libraries in basefp and bezierfloat, which EISC AE32000C emulates.

4. Complexity and Power Consumption

In addition to good performance, our architecture is also attractive in terms of hardware complexity (logic gates) and power consumption. The hardware complexity is measured using Synopsys' Design Compiler and the power consumption is evaluated using the PrimePower tool. We used a statistical activity based on a power analysis with a 0.18 μm CMOS process at 1.62 V supply voltage and at 100 MHz clock frequency. The toggle rate of primary inputs was 50%.

In total, there is a hardware complexity increase of 62.2% (61,185 to 99,215 gates) in the 2-way and 182.9% (to 173,084 gates) in the 4-way MLEP. Power consumption increases by 41.9% (14.624 to 20.749 mW) in the 2-way and by 124.9% (to 32,884 mW) in the 4-way MLEP. Our proposed architecture shows much lower power consumption and complexity than typical dual-core and quad-core architectures [6]. Also, the widened cache interface incurs a complexity overhead of 73.6%; however, this complexity is minimal compared to the total. The difference with and without register duplication is negligible. The resource of the fetch unit in serial and parallel architecture is similar. The hardware complexity of the fetch and decode units was increased by 61.9% in 2-way and 142.9% in the 4-way MLEP. This is due to the following factors: division of the original register file into banks or duplication of the original register file; renaming registers; the addition of special registers (a stack pointer and a status register); the addition of update units, buses and buffer resources; and the addition of a thread controller unit. When more execution units are added, resource utilization increases by 80.6% in the 2-way and 254.9% in the 4-way extensions. The increase in the number of memory and write back units was 62.7% and 226.6% in the 2-way and 4-way MLEP, respectively, (AE32000C combines these two units into one) due to the duplicated buses and signals as well as additional selection units.

V. Related Work

Our architecture is similar to SMT [16] and SIMD execution. The SMT architecture allows multiple threads to compete for and to share all of the processor's resources every cycle; therefore, it converts TLP into ILP [16]. Each thread executes its own instruction streams. The clustered architecture fetches/decodes multiple instructions in one cycle and distributes them in separate instruction queues to clustered functional units [17]. In [18], multiple processors execute independent instruction streams, but in a lockstep manner between cores with an appropriate synchronization through communication. Our architecture uses only one instruction stream which is shared by all cores, and our study demonstrates that the sharing is an acceptable solution in embedded benchmarks.

In SIMD architectures some dedicated instructions are developed to exploit DLP; however, not all instructions in ISA can be used for SIMD execution because some of them do not provide DLP, such as branch, stack push/pop, call instructions, and so on. The limitation incurs a compiler frequently fail to simdize the codes [10]. Also, in order to invoke SIMD execution, data must be moved to SIMD register files, which sometimes incur large overhead. MLEP allows instructions to be executed at the same time by multiple threads like the SIMD execution, but no special data movement is required for parallel execution except for argument passing terminology from a main thread to its child threads. In conventional SIMD

architectures like MasPar [19], no barrier instruction is needed since it does not support control transfer instructions. Our approach is similar to the divergence control and dynamic wrap formation of modern graphics processing units (GPUs) [20]. The GPU approach relies on dynamic instruction wrapping to hide memory latency and to better fill the hardware cores. A stack structure is also used to control the divergence and convergence of threads in our architecture. However, with only a hardware mechanism to keep PCs and their associated information inside the stack as in [20], convergence control is not guaranteed due to the compiler's jump optimization. To resolve this problem, we use an explicit synchronization inside codes by introducing a barrier instruction. Another difference is that we slightly modify the cache interface by widening the cache, which improves spatial locality between threads. The nVidia G80 GPU uses an execution model which is very similar to ours, and uses Parallel Thread Execution (PTX) which is a low-level parallel thread execution virtual machine and ISA for general purpose parallel thread execution. PTX programs are translated at install time to the target hardware instruction set [21]. Also, the G80 GPU uses predicate branches to resolve the divergence control problem [22].

VI. Conclusion and Future Work

In this paper, we introduced a new architecture to execute parallel threads in lockstep by partially duplicating a single pipeline and support from a compiler. The proposed approach uses thread-level parallelism as in an SMT/CMP system, but executes parallel threads in lockstep like a SIMD machine by translating TLP into statically scheduled ILP by a compiler.

Our approach is more favorable than a typical SIMD execution because of its wider range of applicability, much easier and more robust code generation using TLP, the use of the same and all original instruction sets for SIMD execution, no initialization of SIMD registers, and higher parallelism due to parallelization of the outermost loops. More importantly, a programmer can easily annotate parallel loops, which allows a compiler to generate high quality parallel codes. Also, our hardware is much simpler than other architecture variants such as SMT, CMP and in-order superscalar processors due to the support of the compiler. This results in much lower power consumption and resource utilization. The verification and performance evaluation of our proposal was performed by extending a commercial 32-bit embedded core and synthesizing it on Xilinx FPGA.

A major drawback of our approach is that it requires multiple computing resources in duplicated pipeline stages for multiple threads. To solve this problem, we will design an instruction scheduler to use hardware software pipelining (shifting instruction sequences like software pipelining).

References

[1] H.C. Hunter and J.H. Moreno, "A New Look at Exploiting Data Parallelism in Embedded Systems," *CASE*, 2003, pp. 159-169.

[2] I. Karkowski and H. Corporaal, "Exploiting Fine- and Coarse-Grain Parallelism in Embedded Programs," *PACT*, 1998, pp. 60-67.

[3] J.E. Smith and G.S. Sohi, "The Microarchitecture of Superscalar Processors," *Proc. of the IEEE*, vol. 83, Dec. 1995, pp.1609-1624.

[4] D.M. Tullsen et al., "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *ISCA-22*, June 1995.

[5] Analog Devices, Inc. ADSP-BF561 Blackfin Embedded Symmetric Multiprocessor Rev. 0.

[6] ARM. ARM11 MPCore. http://www.arm.com/.

[7] EEMBC (EDN Embedded Microprocessor Benchmark Consortium). http://www.eembc.org.

[8] J. Oh et al., "OpenMP and Compilation Issue in Embedded Applications," *LNCS*, vol. 2716, June 2003, pp. 109-121.

[9] Extendable Instruction Set Computer. http://www.adc.co.kr.

[10] A. Eichenberger et al., "A Tutorial on BG/L Dual FPU Simdization," *BlueGen System Software Workshop*, 2005.

[11] C. Kozyrakis and D. Patterson, "Vector vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks," *MICRO-35*, 2002, pp. 283-293.

[12] D. Talla et al., "Evaluating Signal Processing and Multimedia Applications on SIMD, VLIW, and Superscalar Architectures," *ICCD*, 2000, pp. 163-172.

[13] OpenMP Forum, http://www.openmp.org/. *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*, Oct. 1997.

[14] M. Sato et al., "Design of OpenMP Compiler for an SMP Cluster," *EWOMP*, Sept. 1999, pp. 32-39.

[15] H.G. Nguyen, S.J. Hwang, and S.W. Kim, "Compiler Construction for Lockstep Execution of Multithreaded Processors," *CIT*, 2007, pp. 829-834.

[16] J.L. Lo et al., "Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading," *ACM Trans. Computer Systems*, vol. 15, no. 3, 1997, pp. 322-354.

[17] J. Collins and D. Tullsen, "Clustered Multithreaded Architectures: Pursuing both IPC and Cycle Time," *IPDPS*, 2004, pp. 766-775.

[18] H. Zhong, S.A. Lieberman, and S.A. Mahlke, "Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-Thread Applications, *HPCA*, Feb. 2007, pp. 25-36.

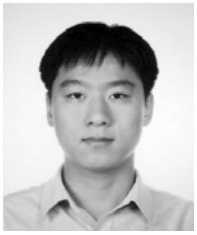
[19] J.R. Nickols, "The Design of the MasPar MP-1: A Cost Effective Massively Parallel Computer," *IEEE COMPCON*, Spring 1990, pp. 25-28.

[20] W.W.L. Fung et al., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," *MICRO*, Dec. 2007, pp. 407-420.

[21] T.R. Halfhill, "Parallel Processing With CUDA," *Microprocessor Report*, Jan. 2008.

[22] GeForce Family, http://www.nvidia.com/page/geforce8.html.

Jaegeun Oh received the BS degree from the School of Electrical Engineering, Korea University, Seoul, Korea, in 2003, and is currently working towards his PhD in the same school. His research interests include processor and SoC design.



Seok Joong Hwang received his BS degree in electrical engineering from Korea University, Seoul, Korea, in 2005, and is working on his PhD in electrical engineering with Korea University. His research interests include microprocessor architecture and compilers.

Huong Giang Nguyen received her BS degree in information technology from Vietnam National University, Hanoi, and she is now working on her PhD in electrical engineering with Korea University, Seoul, Korea. Her research interests include computer system architecture, multithreading, compiler technique, and embedded systems.

Areum Kim received the BE degree in electrical engineering from Korea University, Seoul, Korea, in 2006, and is working on her master's degree in electrical engineering with Korea University. Her research interests include computer system architecture, multithreading and compilers.

Seon Wook Kim received the BS degree from in electronics and computer engineering from Korea University, Seoul, Korea, in 1988. He received the MS degree in electrical engineering from the Ohio State University, Columbus, Ohio, USA in 1990. He received the PhD degree in electrical and computer engineering from Purdue University, West Lafayette, Indiana, USA in 2001. He was a senior researcher at the Agency for Defense Development from 1990 to 1995, and a staff software engineer at Intel/KSL from 2001 to 2002. Currently he is an associate professor with the School of Electrical Engineering of Korea University. His research interests include compiler construction, microarchitecture, and SoC design. He is a senior member of ACM and a member of IEEE.

Chulwoo Kim received the BS and MS degrees in electronics engineering from the Korea University, Seoul, Korea, in 1994 and 1996, respectively, and the PhD degree in electrical and computer engineering from the University of Illinois at Urbana-Champaign, in 2001. Since September 2002, he has been with the Department of Electronics and Computer Engineering, Korea University, where he is currently an assistant professor. His research interests are in the areas of broadband processor design, clocking and latching, low-power/high-performance circuits, and floating-point unit and high-speed I/O. He has authored over 20 papers and one book chapter.

Jong-Kook Kim is currently an assistant professor with Korea University, Seoul, Korea. He received his BS in electronics engineering from Korea University in August 1998. He received his MS and PhD degrees in electrical and computer engineering from Purdue University in May 2000 and August 2004, respectively. He has co-authored 15 technical papers. His research interests include heterogeneous distributed computing, ubiquitous computing, computer architecture, performance measures, resource management, evolutionary heuristics, energy-aware computing, and reliable and collaborative computing. He is a member of the IEEE and ACM.