Study of an In-order SMT Architecture and Grouping Schemes

Byung In Moon, Moon Gyung Kim, In Pyo Hong, Ki Chang Kim, and Yong Surk Lee

Abstract: In this paper, we propose a simultaneous multithreading (SMT) architecture that improves instruction throughput by exploiting instruction level parallelism (ILP) and thread level parallelism (TLP). The proposed architecture issues and completes instructions belonging to the same thread in exact program order. The issue and completion policy greatly reduces the design complexity and hardware cost of our architecture, compared with others that employ out-of-order issue and completion. On the other hand, when the instructions belong to different threads, the issue and completion orders for those instructions may not necessarily be identical to the fetch order. The processor issues instructions simultaneously from multiple threads to functional units by exploiting ILP and TLP, and by dynamic resource sharing. That parallel execution notably improves performance and resource utilization with minimal additional hardware cost over the conventional superscalar processors.

This paper proposes an SMT architecture with grouping as well as one without grouping. Without grouping, all threads dynamically and flexibly share most resources. On the other hand, in the SMT architecture with grouping, in which resources and threads are divided into several groups for design simplification, resources are shared only among threads belonging to the same group as those resources. Simulation results show that our processors with four and eight threads improve performance by three or more times over the conventional superscalar processor with comparable execution resources and policies, and that reasonable grouping reduces the design complexity of SMT processors with little negative effect on performance.

Keywords: Multithreading, SMT, ILP, TLP, in-order issue and completion, grouping.

1. INTRODUCTION

The superscalar processor, the current conventional microprocessor, has arrived at its performance limit, while state-of-the-art IC processes allow more than 100 million transistors on a single chip. This is what directs microprocessor designers to look for the next-generation microprocessor architecture, and many of them are now turning their attention to multithreading [1]. Multithreading hides latency problems by increasing parallelism through TLP, and thus substantially increases processor utilization and significantly improves instruction throughput. Also, unlike VLIW (Very Long Instruction Words) [2], it maintains full binary compatibility.

Multithreading can be classified into one of three categories [1]: coarse, fine, and simultaneous. Coarse multithreading (CMT) supports only one active thread by allowing instructions from only one thread in its execution pipe, and hides only long-latency events such as cache misses. Fine multithreading (FMT) supports multiple active threads, but issues instructions from only one thread in a cycle, and cannot remove horizontal waste [3]. SMT issues and executes instructions from multiple threads each cycle. Among those three categories, SMT outperforms others and is considered as a next-generation architecture. It has no practical processor design due to its great complexity. However, in the near future, processor chips that adopt the SMT architecture are expected to appear on the market. Some companies are known to have adopted this architecture for their next-generation processors [4].

A number of other architectures have been proposed that execute instructions from multiple threads each cycle. Tullsen, et al., [5] propose an architecture

Manuscript received December 3, 2002; revised April 16, 2003; accepted April 23, 2003. This work was supported by the National Research Laboratory Program of the Korea Institute of S & T Evaluation and Planning under grant 2000-N-NL-01-C-246.

Byung In Moon is with the SOC Team, SP Division of System IC, Hynix Semiconductor Inc., Hynix Youngdong Bldg, 891 Daechi-dong, Kangnam-gu, Seoul 135-738, Korea (e-mail: byungin.moon@hynix.com).

Moon Gyung Kim, In Pyo Hong, and Yong Surk Lee are with the Department of Electrical & Electronic Engineering, Yonsei University, 134 Shinchon-dong, Seodaemoon-gu, Seoul 120-749, Korea (e-mail: {bungae, necross}@dubiki.yonsei.ac.kr, yonlee@yonsei.ac.kr).

Ki Chang Kim is with the School of Information & Communication Engineering, Inha University, 253 Yonghyun-dong, Nam-gu, Incheon 402-751, Korea (e-mail: kchang @inha.ac.kr).

for SMT. This architecture is based on register renaming [6-7] and out-of-order issue and completion. Register renaming requires additional registers, which result in a larger register file and a slow down in register file accesses. In addition, the register renaming logic adds an additional stage for register renaming. Moreover, in some instruction set architectures (ISAs), it is almost impossible to support register renaming due to specific architectural features. For example, in the ARM architecture, register renaming is very difficult to implement, since it supports conditional execution for most of its instructions. Also to be noted is the fact that out-of-order completion requires a complicated recovery and restart mechanism for branch misprediction and exception [7]. Such complicated recovery and restart mechanism increases the design complexity of out-of-order processors and causes large penalties in the case of branch misprediction and exception. In combination with the vast complexity of the SMT architecture, these difficulties make the out-of-order SMT impractical even in the near future.

Hirata, et al., [8] present an architecture for a multithreaded superscalar processor. In their architecture, threads do not share instruction queues nor decode slots. Instead, each thread is provided with its own instruction queue and decode slot. Such resource partitioning among threads prevents the instruction queues and decode slots from being shared dynamically. As a result, instruction queues and decode slots are wasted while some threads are unable to fetch and decode instructions. The waste of those resources is particularly more serious when the number of available threads is smaller than that of threads supported by the hardware. Moreover, they do not describe a recovery and restart mechanism for branch misprediction and exception, despite the fact that such a description is necessary due to the out-of-order execution of their architecture.

Several conventional superscalar architectures support out-of-order execution for the purpose of increasing ILP. However, in SMT processors, out-of-order issue and completion policy is unnecessary, because TLP between multiple threads in the SMT architecture compensates for the lack of ILP of each single thread. Thus, we propose an in-order SMT architecture that is derived from the conventional in-order superscalar architecture. Our simple in-order issue and completion policy reduces design complexity and hardware cost to a significant extent compared with out-of-order SMT. Furthermore, we present grouping schemes. Without grouping, all threads dynamically and flexibly share the majority of resources. On the other hand, in the SMT architecture with grouping, in which resources and threads are divided into several groups in order to reduce design complexity, resources are shared only among threads belonging to the same group as those resources.

The rest of the paper is organized as follows. Sect. 2 describes in detail the SMT architecture that we propose as well as the grouping schemes to reduce design complexity and the changes in our SMT architecture from the conventional superscalar architecture. Sect. 3 presents the simulation environment that we constructed, and evaluates the performance of our SMT architecture. We discuss related work in Sect. 4 and summarize our results and conclude the paper in Sect. 5.

2. PROCESSOR ARCHITECTURE

In Sect. 2.1 we propose an SMT architecture in which all threads thoroughly share execution resources. Also, as a method to reduce the design complexity of our SMT architecture, in Sect. 2.2 we present the grouping schemes that divide resources and threads into several groups.

2.1. Processor architecture without grouping

Our processor is derived from the conventional superscalar architecture, and is constructed through applying small changes to the conventional superscalprocessor. As shown in Fig. 1 (the architecture of a four-thread SMT processor), the processor consists of an instruction cache, an instruction memory manage-

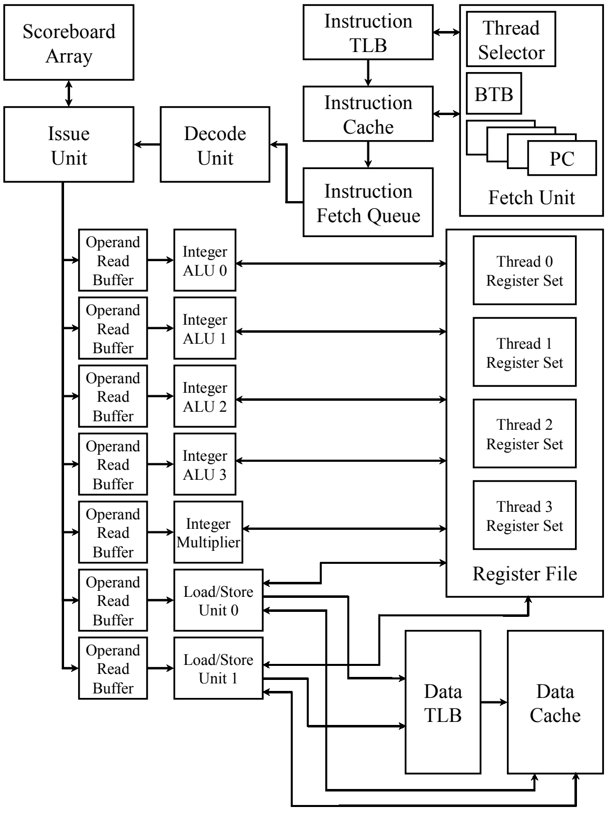


Fig. 1. Overall architecture of the four-thread SMT processor without grouping.

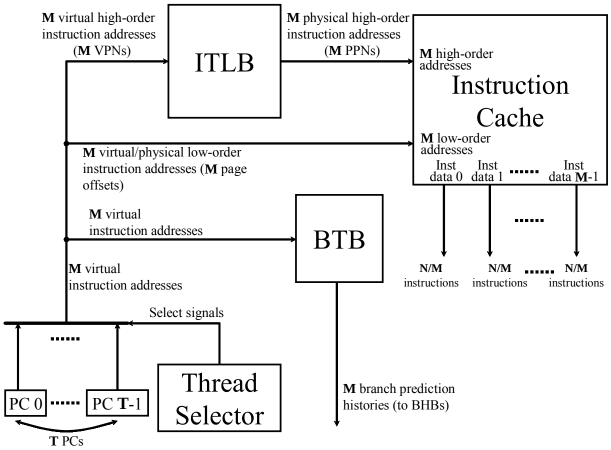

M virtual high-order instruction addresses (M VPNs)
ITLB
M physical high-order instruction addresses (M PPNs)
M high-order addresses
Instruction Cache
M low-order addresses
Inst data 0
Inst data 1
Inst data M-1
M virtual/physical low-order instruction addresses (M page offsets)
M virtual instruction addresses
BTB
M virtual instruction addresses
N/M instructions
N/M instructions
N/M instructions
Select signals
PC 0
PC T-1
T PCs
Thread Selector
M branch prediction histories (to BHBs)

Fig. 2. Instruction fetch structure of the SMT architecture without grouping.

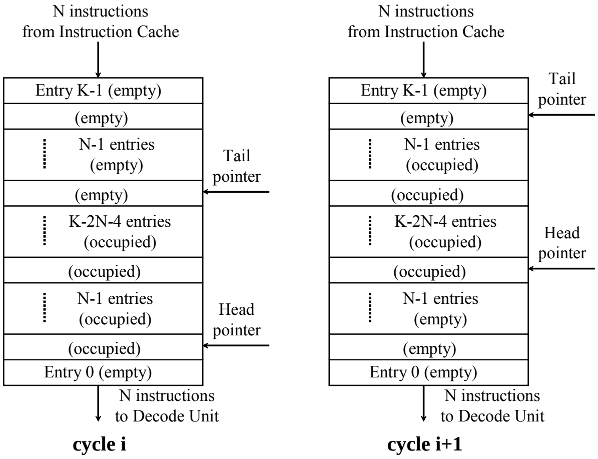


Fig. 3. Instruction fetch queue (a FIFO queue with head and tail pointers).

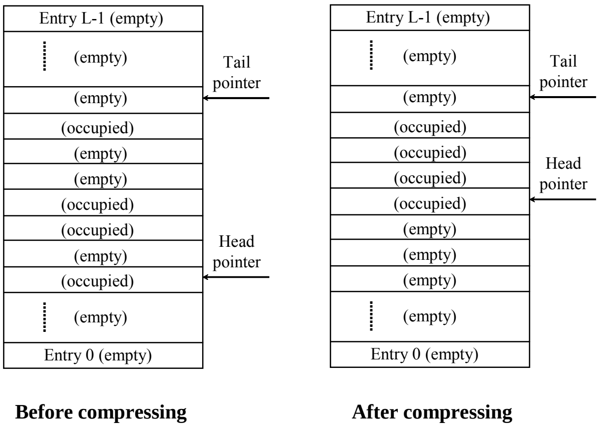


Fig. 4. Instruction issue queue compressing.

ment unit (IMMU), a fetch unit (containing a branch target buffer (BTB), branch history buffers (BHBs), program counters (PCs) and a thread selector), an instruction fetch queue (IFQ), a decode unit, a register file, a scoreboard array, an issue unit (containing an instruction issue queue (IIQ)), functional units (including no floating-point unit), a data cache, and a data memory management unit (DMMU). Most resources in this processor are dynamically shared among threads with the exception of a few resources, as described below.

The fetch unit fetches instructions from multiple threads each cycle. If the fetch width is N instructions and the fetch unit partitions the width among M threads each cycle, the fetch unit fetches N instructions (N/M instructions per thread) from M threads selected from T threads by the thread selector every cycle. The thread selector picks up M threads by a specific fetch priority policy. In this structure, the instruction cache is non-blocking, since, during misses for some threads, the processor continues fetching instructions from other threads. The instruction cache consists of M banks [9] and M ports with the purpose of fetching instructions simultaneously from M threads. The instruction TLB (ITLB) also has M ports in order to translate M virtual addresses from M threads to M physical addresses at the same time. Likewise, the BTB also has M ports, M branch histories from which are saved in the M BHBs related to the selected threads—T separate BHBs exist one per thread. If the number of available threads is smaller than M or bank conflicts occur, then the number of instructions to be fetched becomes less than N instructions. In addition, the number of PCs for fetching instructions must be equal to that of simultaneously supported threads (T) (one PC per thread). Fig. 2 shows the instruction fetch structure.

Fetched instructions are stored in the tail-pointed locations of the IFQ, as shown in Fig. 3, in which the IFQ has K entries. The decode unit decodes instructions in the fetch order from the head-pointed locations of the IFQ. During decoding, the decode unit determines which type of functional unit will execute the decoded instruction. At the same time, the decode unit identifies the addresses of source and destination registers, immediate data, and the operation of each instruction. Finally, those decoded instructions are stored in order in the IIQ.

The issue unit selects instructions from the IIQ by a specific issue priority policy and examines their data dependencies and resource conflicts. Then, instructions without data dependencies or resource conflicts are issued to their assigned functional units. The issue unit issues in order instructions belonging to the same thread. However, the issue order of instructions from different threads may not be the same as the fetch order. Besides, the issue unit supports per-thread instruction flush. Those two factors cause empty (invalid) entries to come into being between the head-pointed entry and the tail-pointed entry. Therefore, the IIQ compressing shown in Fig. 4, in which IIQ has L entries, is necessary and performed simultaneously with the instruction issue. The scoreboard array, which has as many entries as registers in the register file, traces the recent update of each register, and provides the issue unit with the information about data dependency and bypassing of each register. On the other hand, the issue unit updates the scoreboard entries corresponding to the destination registers of issued instructions.

Issued instructions obtain their operands through bypassing, or read the operands from the register file containing the register sets for all of the threads. Then, they are executed in their assigned functional units. The architecture provides three types of functional units: arithmetic logic unit (ALU), multiply unit, and load/store unit (LSU). Fig. 5 shows the operations of the functional units. As shown in Fig. 5, the functional units execute instructions issued to them, and write the results to the register file. The processor completes instructions in order, and thus, the writes to the register file are performed in order, for each thread.

The SMT architecture improves instruction throughput by exploiting TLP as well as ILP and converting TLP to ILP, so that the TLP of the SMT architecture can compensate for the ILP loss caused by employing in-order execution rather than out-of-order execution. For example, if the number of supported threads and the instruction issue width are 8 threads and 4 instructions, respectively, the ILP of 0.5 instructions per cycle (IPC) per thread is sufficient to fully utilize the instruction issue width of 4 instructions. For this reason, it is unnecessary to employ out-of-order execution in the SMT architecture. Actually, the simulation results in Sect. 3 show that despite in-order execution, IPC is almost the same as instruction issue width in case that the number of supported threads is greater than the issue width.

In-order execution in this paper reduces design complexity and hardware cost as compared with out-of-order execution. In this architecture, recovering the in-order state of the processor from mispredicted branches and exceptions is very simple thanks to its in-order completion feature, and is performed only by flushing the instructions of the corresponding thread that follow the mispredicted branch or excepted instruction. Thus, in our SMT architecture, the additional complexity for recovering from branch misprediction and exception is only a per-thread instruction flush mechanism compared with the conventional superscalar architecture. On the other hand, the SMT architecture employing out-of-order execution necessitates additional registers for renaming that increase the register file area and slow down register file accesses, an extra pipeline stage for renaming, which increases branch misprediction penalty, and a register renaming logic. Also, it needs storage and logic units to maintain in-order, lookahead and architectural states [7]. Moreover, the complex mechanism for recovering and restarting from branch misprediction and exception greatly increases the design complexity of the SMT architecture employing out-of-order execution.

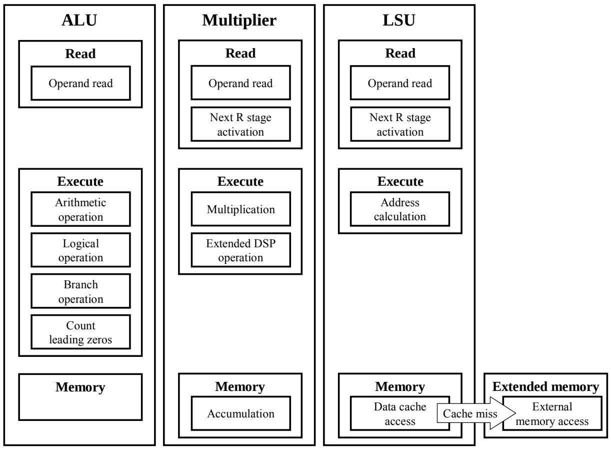


Fig. 5. Operations of functional units in the stages R, E, and M.

2.2. Grouping

By sharing the majority of hardware resources among all threads, an SMT processor notably improves resource utilization and performance. In that case, however, the design complexity of the SMT processor becomes too large. In this subsection, grouping schemes are proposed as methods to decrease the design complexity of our SMT architecture. Figs. 6 and 7 show the structures that read register operands and send them to ALUs. In Fig. 7, the read ports, ALUs and threads are divided into two groups, but not in Fig. 6. By grouping, the structure multiplexing register addresses and ALU operands in Fig. 7 is simpler than that in Fig. 6.

Table 1. Three grouping schemes and grouped resources in each scheme.

Grouping scheme	Grouped resources
GRP 1	Instruction cache ports. Instruction TLB ports. BTB ports.
GRP 2	Instruction fetch queues. Decode slots. Instruction issue queues.
GRP 3	Instruction issue control logic. Read and write ports of the register file. Read and write ports of the scoreboard arrays. Functional units. Data cache ports. Data TLB ports.

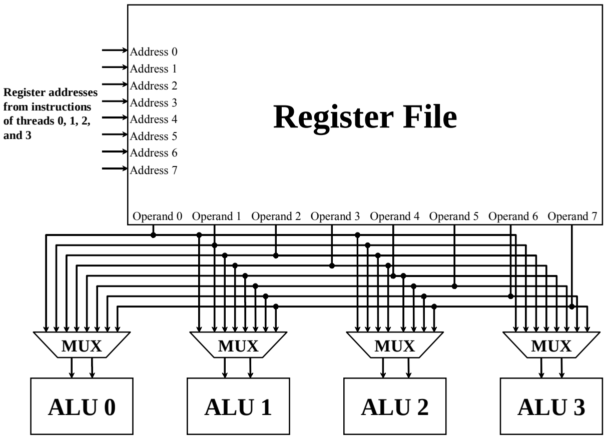


Fig. 6. Register reads without grouping.

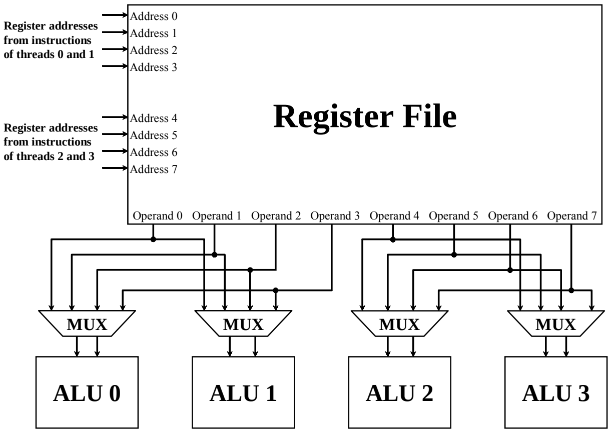


Fig. 7. Simplification of register reads by grouping.

The above grouping can be optionally applied to all hardware resources, and we present three grouping schemes: GRP 1, GRP 2 and GRP 3, depending on the types of grouped resources, as shown in Table 1. The three schemes can be applied independently of each other, since each scheme is applied to its own exclusive hardware resources as shown in Fig. 8. By grouping, the resources of each group are shared only among the threads belonging to the same group. For example, if GRP 1 divides resources for the instruction fetch including instruction cache ports, instruction TLB ports and BTB ports into two groups (the tions using only the fetch resources of the same group. Grouping threads and related resources into two groups by GRP 1 is shown in the instruction fetch structure of Fig. 9. A similar grouping mechanism is applied to GRP 2 and GRP 3.

Hardware design complexity is reduced by the above grouping schemes as described below.

- ● By GRP 1, the thread selection logic for fetch is simplified.
- ● By GRP 1, virtual address paths to the ITLB and BTB are simplified.
- ● By GRP 1, branch prediction history paths from the BTB to the BHB are simplified.
- ● By GRP 1, the ITLB part can be implemented as small one-read-port TLBs (one TLB per group), instead of one large multi-read-port TLB.
- ● By GRP 1, the instruction cache part can be implemented as small one-read-port caches (one cache per group), instead of one large multi-read-port cache.
- ● By GRP 1, the BTB part can be implemented as small one-read-port BTBs (one BTB per group), instead of one large multi-read-port BTB.
- ● By applying GRP 1 and GRP 2 in combination, instruction paths from the instruction cache to the IFQ are simplified.
- ● By GRP 2, management of the IFQ and IIQ is simplified.
- ● By GRP 3, the register file part is implemented as small register files (with fewer ports), instead of one large register file.
- ● By GRP 3, the instruction issue control logic (e.g., data dependency check, thread selection for issue, and functional unit assignment) is greatly simplified.
- ● By GRP 3, register address paths to the register file are simplified.
- ● By GRP 3, register data paths from and to the register file are simplified.
- ● By GRP 3, data bypassing logic is simplified.

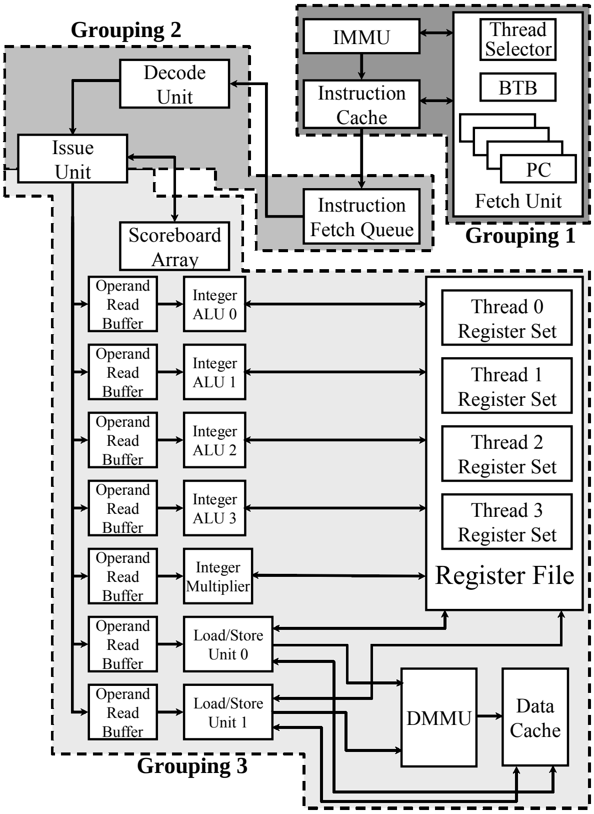


Fig. 8. Three grouping schemes in the in-order SMT architecture.

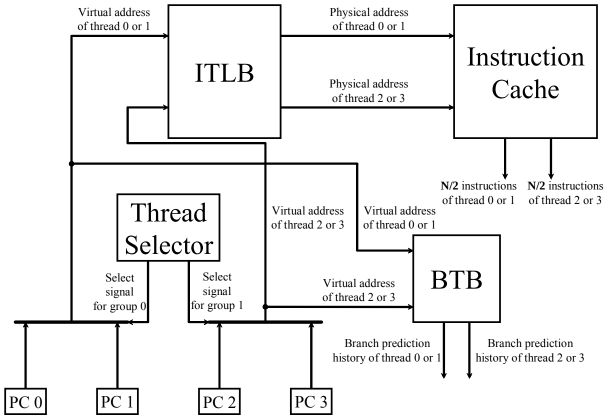

Virtual address of thread 0 or 1
ITLB
Physical address of thread 0 or 1
Physical address of thread 2 or 3
Instruction Cache
N/2 instructions of thread 0 or 1
N/2 instructions of thread 2 or 3
Thread Selector
Virtual address of thread 2 or 3
Virtual address of thread 0 or 1
Virtual address of thread 2 or 3
BTB
Select signal for group 0
Select signal for group 1
Branch prediction history of thread 0 or 1
Branch prediction history of thread 2 or 3
PC 0
PC 1
PC 2
PC 3

Fig. 9. Instruction fetch structure with the grouping scheme GRP 1.

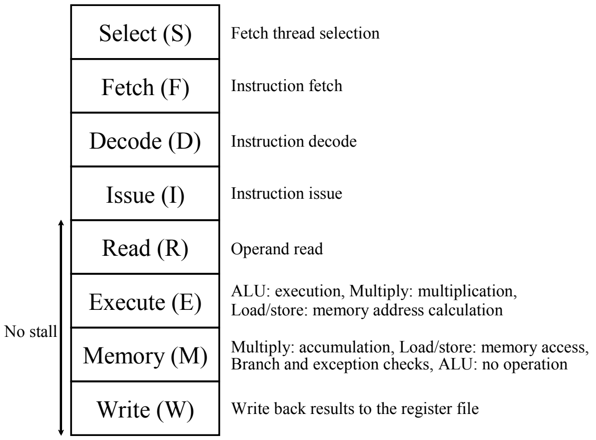
- By GRP 3, the data TLB (DTLB) part can be implemented as small one-read-port TLBs (one TLB per group), instead of one large multi-read-port TLB.
- By GRP 3, the data cache part can be implemented as small one-port caches (one cache per group), instead of one large multi-port cache.

2. 3. Pipeline structure

The pipeline structure shown in Fig. 10 is applied to our SMT architecture regardless of grouping, and is summarized below.

- In the *select* (S) stage, the thread selector picks up threads from which instructions are to be fetched in the next cycle.
- In the *fetch* (F) stage, instructions are fetched from threads selected in the select stage and stored in the IFQ.
- In the *decode* (D) stage, instructions are decoded, and then the decoded instructions are stored in the IIQ.
- In the *issue* (I) stage, instructions in the IIQ without data dependencies or resource conflicts are issued to the functional units.
- In the *read* (R) stage, instruction operands are read from the register file or bypassed.
- In the *execute* (E) stage, ALU instructions are executed in the ALUs, and the multiply instructions perform multiplication. In the case of load/store instructions, memory addresses are calculated.
- In the *memory* (M) stage, the multiply instructions perform accumulation, and load/store instructions access the data cache. Also, branch misprediction and exception checks are carried out. ALU instructions do not carry out any operation in this stage.
- In the *write* (W) stage, execution results are written back to the register file.

In addition, there is no instruction stall from the R stage to the W stage, because the issue unit arbitrates instruction issues to prevent those stages from stalling.



Select (S)	Fetch thread selection
Fetch (F)	Instruction fetch
Decode (D)	Instruction decode
Issue (I)	Instruction issue
Read (R)	Operand read
Execute (E)	ALU: execution, Multiply: multiplication, Load/store: memory address calculation
Memory (M)	Multiply: accumulation, Load/store: memory access, Branch and exception checks, ALU: no operation
Write (W)	Write back results to the register file

Fig. 10. Pipeline structure of the SMT processor.

2.4. Changes from superscalar architecture

While the SMT architecture has excellent potential to increase processor performance, it can add substantial complexity to the design. First, throughout the pipeline stages, each instruction is accompanied by its thread identifier. The following changes are made to support multiple threads simultaneously.

- An instruction cache that consists of several banks and is non-blocking.
- Multiple PCs and a thread selector that picks up threads from which instructions are to be fetched.
- A fetch priority policy and an issue priority policy.
- A thread id with each BTB entry and a separate branch history buffer for each thread for predicting branches with higher accuracy.
- Multiple IFQs and IIQs if the grouping scheme GRP 2 is applied.
- IIQ compressing.
- Per-thread instruction dependency check.
- Per-thread branch misprediction check.
- Per-thread exception mechanism.
- Per-thread instruction flush mechanism.
- The register file T (the number of threads) times larger than that of the conventional superscalar processor.
- The scoreboard array T times larger than that of the conventional superscalar processor.

3. SIMULATION AND RESULTS

Our method of performance evaluation is based on execution-driven simulation. An execution-driven, cycle-based simulator written in C language is implemented to model in detail our SMT architecture. The simulator adopts the ARM architecture as its ISA, and has a variety of configuration parameters in order to simulate SMT processors with various organizations as shown in Table 2. The fetch width and decode width are the same as the issue width, and thus not listed in Table 2. The bank conflicts [9] of the instruction and data caches are ignored in the simulation. Thus, the parameters for the numbers of the banks of the caches are meaningless in the simulator, and not listed in Table 2. The parameters FETCH_PRI and ISSUE_PRI of Table 2 select fetch and issue priority policies respectively, and the selectable policies are summarized in Table 3.

Table 2. Configuration parameters.

Configuration parameter	Parameter description
GRP1_NUM	Number of GRP 1 groups
GRP2_NUM	Number of GRP 2 groups
GRP3_NUM	Number of GRP 3 groups
THREAD_NUM	Number of supported threads
ISSUE_WIDTH	Instruction issue width
IFQ_SIZE	Number of IFQ entries
IIQ_SIZE	Number of IIQ entries
ALUF_NUM	Number of ALUs
MULF_NUM	Number of multiply units
LSUF_NUM	Number of LSUs
BTB_SIZE	Number of BTB entries
ITLB_SIZE	Number of ITLB entries
DTLB_SIZE	Number of DTLB entries
ICP_NUM	Number of instruction cache ports
IC_SIZE	Instruction cache size
ICWAY_NUM	Instruction cache associativity
ICBLK_SIZE	Block size of the instruction cache
DCP_NUM	Number of data cache ports
DC_SIZE	Data cache size
DCWAY_NUM	Data cache associativity
DCBLK_SIZE	Block size of the data cache
CFILL_CYC	Number of cycles spent refilling or evicting one cache block
FETCH_PRI	Fetch priority policy
ISSUE_PRI	Issue priority policy

Our workload is from the SPEC CPU2000 benchmark suite [10] and the ADS (ARM Developer Suite) [11]. Table 4 shows the benchmark programs that we choose from those suites, as well as their brief descriptions. Each thread of the simulator executes the six benchmark programs in Table 4. We adjust the execution orders of programs differently depending on threads, for the purpose of preventing abnormal interthread cache interference, which occurs while multiple threads execute the same program simultaneously. In this way, we model a parallel workload achieved by multiprogramming rather than parallel processing. Thus, performance results are not affected by synchronization delays and inefficient parallelization.

Fig. 11 shows the flow of simulation and performance evaluation. First, we ensure that the simulator has the proper organization by adjusting configuration parameters and compiling the source of the simulator written in C language. And we obtain ARM executable binary files of ELF format by compiling each benchmark program with the built-in compiler of the ADS. Then, the simulator performs simulations using those executable files. Finally, we evaluate the performance of SMT processors using the performance data that the simulator outputs at the end of simula-

Table 3. Fetch and issue priority policies.

Type of priority policy	Priority policy	Description
Fetch	RR	Select threads using a round-robin priority scheme
	ICOUNT_IFQ	Give highest priority to those threads with the fewest instructions in the IFQ(s)
	ICOUNT_Q	Give highest priority to those threads with the fewest instructions in the IFQ(s) and IIQ(s)
	ICOUNT_ALL	Give highest priority to those threads with the fewest instructions in the IFQ(s), IIQ(s) and functional units
	ICOUNT_BR	Give highest priority to those threads with the fewest unresolved branches
	ICOUNT_MISS	Give highest priority to those threads that have the fewest outstanding data cache misses
	IIQOL	Give lowest priority to those threads with instructions closest to the head(s) of the IIQ(s)
Issue	OLDEST	Give highest priority to those instructions closest to the head(s) of the IIQ(s)
	ICNT_FU	Give highest priority to instructions belonging to those threads with the fewest instructions in functional units
	ICNT_MISS	Give highest priority to instructions belonging to those threads that have the fewest outstanding data cache misses

Table 4. Benchmark programs used to evaluate the performance of SMT processors with various configurations.

Benchmark suite	Program	Program description
SPEC CPU2000	parser	Syntactic parser of English, based on link grammar
	twolf	TimberWolfSC placement and global routing package
	vortex	Single-user object-oriented database transaction benchmark
	mcf	Program designed for the solution of single-depot vehicle scheduling problems
ADS	sorts	Benchmark performing and comparing insertion, shell and quick sorts
	Dhrystone	Benchmark used to measure the integer processing performance of a system

tions. In performance evaluation, IPC is used as a performance criterion.

Simulations are carried out for evaluating the performance of the SMT processors without grouping. Figs. 12 and 13 are the results of those simulations with configuration parameter values of Tables 5 and 6, respectively. As shown in Fig. 12, as long as the number of threads is less than or equal to four, the performance of four-issue SMT processors remarkably improves as the number of threads increases. Plus, Fig. 13 shows that increasing the number of threads highly improves the performance of eight-issue SMT processors as long as the number of threads is less than or equal to eight. Putting the results of Figs. 12 and 13 together, we discover the fact that SMT processors are cost-effective when the number of threads is equal to that of instructions that can be issued in one cycle. The poor performance of single-threaded (superscalar) processors results from the low ILP feature specific to the ARM ISA. However, SMT processors with four and eight threads improve performance by three and five times, respectively, over single-threaded processors. This means that processors adopting ISAs having the feature of low ILP can benefit more from multiple threads and that the ARM is one of those ISAs. In addition, as shown in Figs. 12 and 13, the performance of SMT processors with eight threads and eight-way caches is better than with sixteen threads and four-way caches. This signifies that increasing the number of cache ways from four to

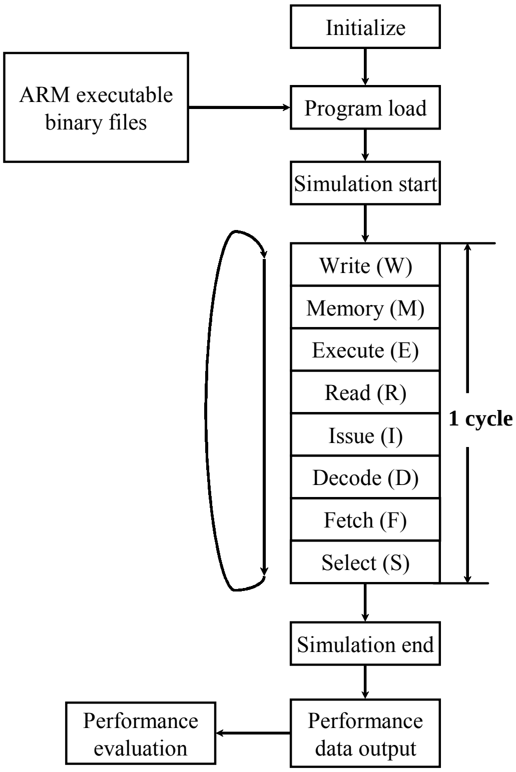


Fig. 11. Flow of simulation and performance evaluation.

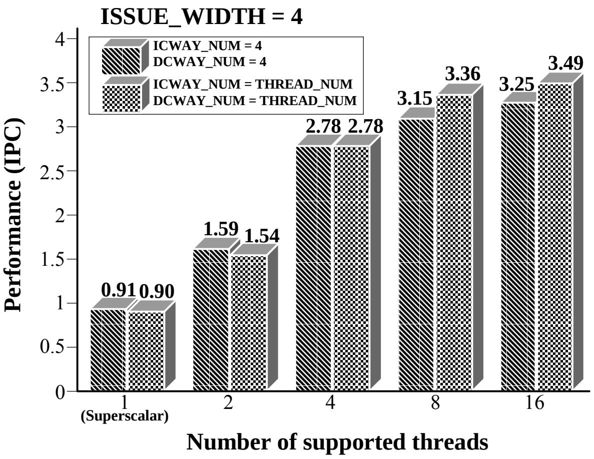


Fig. 12. Performance comparison of four-issue processors with parameter values given in Table 5.

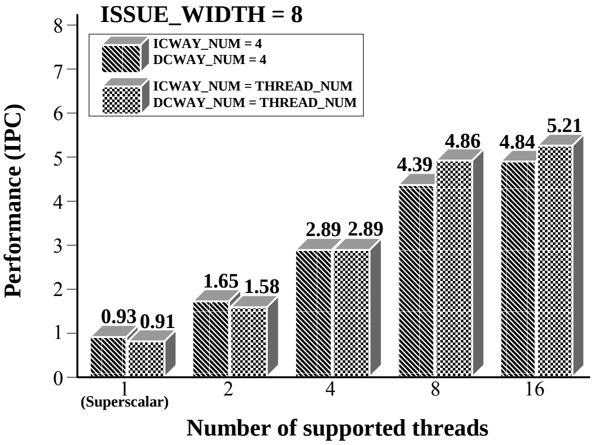


Fig. 13. Performance comparison of eight-issue processors with parameter values given in Table 6.

Table 5. Values of configuration parameters of four-issue processors (for the performance comparison in Fig. 12).

Configuration parameter	Parameter value
ISSUE_WIDTH	4
IFQ_SIZE	32
IIQ_SIZE	32
ALUF_NUM	4
MULF_NUM	Equal to GRP3_NUM
LSUF_NUM	2
BTB_SIZE	64
ITLB_SIZE	64
DTLB_SIZE	64
ICP_NUM	1 for superscalar 2 for 2-thread or 4-thread SMT 4 for 8-thread or 16-thread SMT
IC_SIZE	64 Kbytes
ICBLK_SIZE	32 bytes
DCP_NUM	2
DC_SIZE	64 Kbytes
DCBLK_SIZE	32 bytes
CFILL_CYC	5
FETCH_PRI	RR
ISSUE_PRI	OLDEST

Table 6. Values of configuration parameters of eight-issue processors (for the performance comparison in Fig. 13).

Configuration parameter	Parameter value
ISSUE_WIDTH	8
IFQ_SIZE	64
IIQ_SIZE	64
ALUF_NUM	8
MULF_NUM	Equal to GRP3_NUM
LSUF_NUM	4
BTB_SIZE	64
ITLB_SIZE	64
DTLB_SIZE	64
ICP_NUM	1 for superscalar 2 for 2-thread or 4-thread SMT 4 for 8-thread or 16-thread SMT
IC_SIZE	64 Kbytes
ICBLK_SIZE	32 bytes
DCP_NUM	4
DC_SIZE	64 Kbytes
DCBLK_SIZE	32 bytes
CFILL_CYC	5
FETCH_PRI	RR
ISSUE_PRI	OLDEST

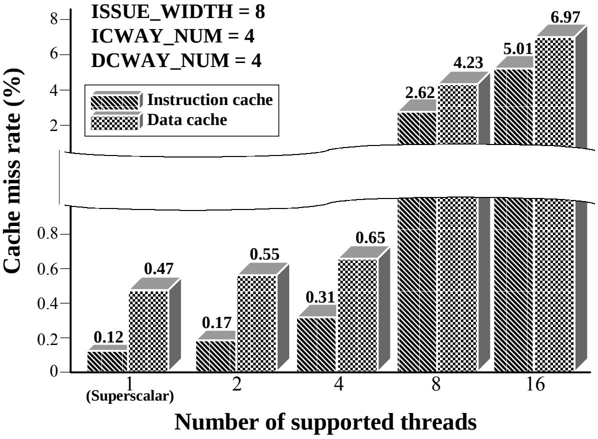


Fig. 14. Cache miss rates of four-way set-associative instruction and data caches (for 1, 2, 4, 8 and 16 threads).

eight improves performance to a greater extent than increasing that of threads from eight to sixteen. This is because as the number of threads increases over that of cache ways, the number of inter-thread conflict cache misses rapidly rises, and as a result, miss rates highly increase, as shown in Fig. 14.

Table 7 shows the performance comparison of various fetch and issue priority policies through the results of simulations that are performed using eight-thread, eight-issue SMT processors with parameter values of Table 6 and eight-way caches. The ICOUNT_Q outperforms the other fetch priority policies and increases performance by 4% compared with the RR. Furthermore, the policy ICOUNT_Q is very simple, considering it can be implemented merely by counting the instructions of each thread in the IFQ and IIQ. Thus, the ICOUNT_Q is a highly recommended policy for our SMT architecture. All the three issue priority policies show almost identical performance.

The grouping in Sect. 2.2 simplifies the design of the SMT processors, but decreases performance. Table 8 shows grouping effects on the performance of eight-thread, eight-issue SMT processors with parameter values of Table 6 and eight-way caches. Applying the GRP 1 grouping scheme simplifies the design of the logic related to the instruction fetch only with trivial (less than 1%) performance decrease compared with the SMT processor without grouping, as shown in Table 8. On the other hand, the performance decrease due to the GRP 3 grouping scheme is relatively large. Thus, a large number of GRP 1 groups and a small number of GRP 3 groups are recommended. Also, moderate grouping is, on the whole, highly recommended, since it reduces the design complexity of the SMT processors with little negative effect on performance.

Table 7. Performance comparison of fetch and issue priority policies.

Type of priority policy	Priority policy	Performance (IPC)
Fetch (FETCH_PRI) (ISSUE_PRI = OLDEST)	RR	4.86
	ICOUNT_IFQ	5.02
	ICOUNT_Q	5.07
	ICOUNT_ALL	5.06
	ICOUNT_BR	4.92
	ICOUNT_MISS	4.94
	IIQOL	5.03
Issue (ISSUE_PRI) (FETCH_PRI = RR)	OLDEST	4.86
	ICNT_FU	4.88
	ICNT_MISS	4.85

Table 8. Grouping effects on the performance of eight-thread, eight-issue SMT processors.

Number of GRP 1 groups	Number of GRP 2 groups	Number of GRP 3 groups	Performance (IPC)
1	1	1	4.86
2	1	1	4.84
1	2	1	4,80
1	1	2	4.77
4	1	1	4.83
1	4	1	4.71
1	1	4	4.57
2	2	2	4.76
4	2	2	4.74
2	4	2	4.67
2	2	4	4.52
4	4	2	4.65
2	4	4	4.47
4	2	4	4.50
4	4	4	4.46

4. RELATED WORK

Tullsen et al. [3] present an idealized SMT architecture model to evaluate the performance potential of simultaneous multi-thread instruction issue. Through use of a multiprogrammed workload of scientific applications, the study demonstrates that superscalar and fine-grain multithreaded processors lack sufficient ILP to fully utilize functional units in a wide-issue processor. By allowing multiple threads to issue each cycle, instruction throughput increased greatly. A later study [5] presents an implementable SMT microarchitecture. This study evaluates several fetch and issue policies for improving the performance of an SMT processor and achieves a 2.5 speedup over a single-threaded superscalar architecture. However, this architecture adopts out-of-order execution, and thus its design complexity is very high compared with our architecture. Also, our grouping schemes proposed as methods to reduce the design complexity of the SMT architecture are not investigated in this study.

Hirata et al. [8], Yamamoto and Nemirovsky [12], and Prasadh and Wu [13] propose architectures that dynamically pack instructions from different threads. In all three architectures, however, the early stages of the pipeline are partitioned. Hirata's architecture includes a separate instruction queue and a fetch/decode unit for each thread. When an instruction is ready, the decode unit must arbitrate for an available functional unit. If there are no available functional units, then the ready instruction is placed in a standby station. This architecture achieves a 3.5 speedup over a single-threaded superscalar architecture when using eight parallel threads for a ray-tracing application. Yamamoto and Nemirovsky propose a similar architecture, where each thread has its own 8-entry dispatch window and fetch unit. Ready instructions compete for available functional units. Prasadh and Wu also require per-thread register sets and instruction fetch units. Although these three architectures permit flexible sharing of processor functional units, they require replicated fetch/decode units and partition the dispatch window/instruction queue. A partitioned instruction queue in these architectures can lead to under-utilization of this resource. On the other hand, our SMT architecture shares the instruction queue between all threads, or shares a partitioned instruction queue between the threads belonging to the same group as the queue by the GRP 2 grouping scheme, so that our paper investigates the trade-offs of resource sharing. Moreover, the instruction queues in these three architectures are fed by the separate fetch units, so that more instruction cache ports are required than in our SMT architecture using a single fetch unit.

The above architectures did not consider the grouping schemes and did not investigate their effects on the performance of the SMT architecture. By sharing most of the hardware resources among all threads, an SMT processor notably improves resource utilization and performance. In that case, however, the design complexity of the SMT processor becomes too large. Thus, our grouping schemes must be investigated when designing practical SMT processors. Additionally, because these architectures focus on increasing performance, they support out-of-order execution for each thread. However, in-order execution for each thread in our architecture shows sufficient performance compared with these out-of-order SMT architectures, as shown in the simulation results.

5. CONCLUSIONS

In this paper, we have proposed an SMT architecture that issues and completes instructions in order, for each thread. Our in-order issue and completion architecture reduces design complexity and hardware cost while maintaining competitive or higher performance compared with the previous SMT architectures using out-of-order execution. It improves processor utilization by dynamic resource sharing among threads. Moreover, our architecture is derived from applying small changes to the conventional in-order superscalar microprocessors. The simulation results show that our processors with four and eight threads improve performance by three or more times over the conventional superscalar processor with comparable resources and policies, and that moderate grouping reduces the design complexity of the SMT processors with little negative effect on performance. Besides, the number of cache ways equal to or over that of supported threads is recommended in the SMT architecture.

REFERENCES

[1] P. Song, "Multithreading comes of age," *Microprocessor Report*, vol. 11, no. 9, July 14, 1997.

[2] A. Abnous and N. Bagherzadeh, "Special features of a VLIW architecture," *Proc. 5th International Parallel Processing Symposium*, pp. 224-227, Anaheim, California, April 1991.

[3] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," *Proc. 22nd International Symposium on Computer Architecture*, pp. 392-403, Santa Margherita Ligure, Italy, June 1995.

[4] K. Diefendorff, "Compaq chooses SMT for Alpha," *Microprocessor Report*, vol. 13, no. 16, December 6, 1999.

[5] D. M. Tullsen, S. J. Eggers, J. S. Emer, H.M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor," *Proc. 23rd International Symposium on Computer Architecture*, pp. 191-202, Philadelphia, Pennsylvania, May 1996.

[6] D. A. Patterson and J. L. Hennesy, *Computer Architecture: A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, San Francisco, California, 1996.

[7] M. Johnson, *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[8] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa, "An elementary processor architecture with simultaneous instruction issuing from multiple threads," *Proc. 19th International Symposium on Computer Architecture*, pp. 136-145, Queensland, Australia, May 1992.

[9] G. S. Sohi and M. Flanklin, "High-bandwidth interleaved memories for vector processors-a simulation study," *IEEE Trans. Comput.*, vol. 42, no. 1, pp. 34-44, January 1993.

[10] J. L. Henning, "SPEC CPU2000: measuring CPU performance in the New Millennium," *IEEE Computer*, vol. 33, no. 7, pp. 28-35, July 2000.

[11] *ARM Developer Suite: Compiler, Linker, and Utilities Guide*, ARM Limited, 2000.

[12] W. Yamamoto and M. Nemirovsky. "Increasing superscalar performance through multistreaming," *Proc. IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, pp. 49–58, June 1995.

[13] R. Prasadh and C.-L. Wu, "A benchmark evaluation of a multi-threaded RISC processor architecture," *Proc. International Conference on Parallel Processing*, vol. 1, pp. 84–91, August 1991.

Byung In Moon received his B.S. and M.S. degrees in Electronic Engineering from Yonsei University, Seoul, Korea, in 1995 and 1997, respectively, where he also received his Ph.D. degree in electrical and electronic engineering. His doctoral research encompasses the subject of SMT architecture design for next-generation high-performance microprocessors. In 2002, he joined the SOC Team, System IC, Hynix Semiconductor Inc. in Korea, where he is currently a design engineer. His research fields include high-performance parallel computer architectures, embedded DSP processors, network processors, and microprocessors.

Moon Gyung Kim received his B.S. and M.S. degrees in Electronic Engineering from Yonsei University, Seoul, Korea in 1997 and 1999, respectively, where he is currently pursuing his Ph.D. degree in electrical and electronic engineering. His current research area includes design and modeling of microprocessor architectures, tures, network processors, and VLSI circuits.

In Pyo Hong received his B.S. and M.S. degrees from Yonsei University, Seoul, Korea in 1999 and 2001, respectively, both in electronic engineering. He is currently working towards his Ph.D. degree in electrical and electronic engineering at the same institution. His current research area comprises next-generation high-performance microprocessor architectures, high-speed computational units, and VLSI circuit design.

Ki Chang Kim received his B.S. degree in Computer Science from California Polytechnique University, Pomona, in 1986, and his M.S. and Ph.D. degrees in Computer Science from the University of California, Irvine, in 1988 and 1992, respectively. He worked as a post-doctor in the IBM T.J. Watson research center, Yorktown Heights, for two years immediately following graduation. In 1994, he joined the faculty of the School of Information & Communication Engineering, Inha University in Korea, where he is currently an associate professor. His current research interests are in parallelizing compilers, performance of network processors, and real-time embedded systems.

Yong Surk Lee received his B.S. degree from Yonsei University, Seoul, Korea, in 1973, and his M.S. and Ph.D. degrees from the University of Michigan at Ann Arbor, in 1977 and 1981, respectively, all in electrical engineering. He worked in the field of microprocessor design in Silicon Valley from 1982 to 1992. Since 1993, he has been a Professor in the Department of Electrical & Electronic Engineering, Yonsei University, Seoul, Korea. His current research interests are in computer architectures, microprocessors, and network processors.