A Simultaneous Multithreading Processor Architecture with Minimal Hardware Overhead

Byung In Moon, Moon Gyung Kim, Woo Kyeong Jeong, Seung Pil Lee, and Yong Surk Lee
Processor Lab., Dept. of Electrical and Electronic Eng., Yonsei Univ.
134 Shinchon-dong, Seodaemoon-gu, Seoul 120 749, Korea
Tel: +82-2-2123-2872, Fax: +82-2-312-4584
E-mail: blue@dubiki.yonsei.ac.kr

Abstract: In this paper, we propose a multithreaded processor architecture that improves instruction throughput by exploiting instruction level parallelism (ILP) and thread level parallelism (TLP). The processor to be proposed issues and completes in order instructions belonging to the same thread. These issue and completion policies greatly reduce design complexity and hardware cost of our architecture, compared with the one adopting out-of-order issue and completion. On the other hand, when the instructions belong to the different threads, the issue and completion orders for those instructions may not necessarily be the same with the fetch order. In this architecture, dynamically and flexibly are shared most resources, which include an instruction cache, an instruction TLB, a fetch unit, a decode unit, read and write ports of a register file, an instruction issue queue, functional units, a data cache, a data TLB, and result buses. The processor issues simultaneously instructions from multiple threads to functional units by exploiting ILP and TLP and by dynamic resource sharing. This parallel execution notably improves performance and resource utilization with minimal additional hardware cost, over the conventional superscalar processors. Simulation results show that our processors with four and eight threads improve performance by two or more times over the conventional superscalar processor with comparable resources and policies.

1. Introduction

Superscalar processor, the main stream of the current microprocessors, arrives at its performance limit, while new IC processes allow more than 10 million transistors on a chip. This is what leads microprocessor designers to look for the next-generation microprocessor architecture, and many of them are now giving attention to multithreading. Multithreading hides latency problems by increasing parallelism through TLP, and thus substantially increases processor utilization and significantly improves instruction throughput. Also, unlike VLIW, it maintains full binary compatibility.

Multithreading can be classified into one of three categories: coarse, fine, and simultaneous. Coarse multithreading 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 supports multiple active threads, but issues instructions from only one thread in a cycle, and cannot remove horizontal wastes [1]. Simultaneous multithreading issues and executes instructions from multiple threads each cycle. Among these three categories, simultaneous multithreading outperforms others and is considered as the next-generation architecture. It has no practical processor design due to its large complexity. However, in the near future, a processor chip that adopts the simultaneous multithreading architecture will appear in the market. Actually, Compaq has officially adopted this architecture for Alpha 21464 [2].

A number of other architectures have been proposed that execute instructions from multiple threads each cycle. Tullsen, et al., [3] propose an architecture for simultaneous multithreading. However, this architecture is based on register renaming and out-of-order issue and completion. Register renaming needs additional registers, which results in a larger register file and makes access to the register file very slow. In addition, the register renaming logic adds one more stage for register renaming. Moreover, in some architectures, register renaming is almost impossible to support due to specific architectural features. For example, in the ARM architecture, register renaming is very difficult since it supports conditional execution for almost all instructions. Also to be noted is the fact that out-of-order completion requires complicated recovery and restart mechanism for branch misprediction and exception. Such complicated recovery and restart mechanism increases design complexity of out-of-order processors and causes large penalty in case of branch misprediction and exception. These difficulties make the out-of-order simultaneous multithreading impractical even in the near future.

Hirata, et al., [4] present an architecture for a multithreaded superscalar processor. In their architecture, threads do not share a single instruction queue unit nor decode unit. Instead, it provides each thread with its own instruction queue unit and decode unit. Such fixed partitioning among threads prevents the instruction queue and decode units from being shared dynamically. As a result, instruction queue and decode units are wasted while some threads are not able to fetch and decode instructions. This is particularly more so, when the number of available threads is smaller than that of hardware threads. Moreover, they do not describe recovery and restart mechanism for branch prediction and exception, despite the fact that such description is necessary due to out-of-order execution of their architecture.

Our architecture supports dynamic sharing of resources including a single fetch queue and decode unit, and thus minimizes resource waste and greatly improves utilization of functional units. In addition, our simple in-order issue

and completion algorithms reduce design complexity and hardware cost to a significant extent.

2. Processor Architecture

2.1 Overall Processor Architecture

Our processor is derived from the conventional superscalar architecture, and is constructed through applying small changes to the conventional superscalar processor. As shown in Fig. 1 (the architecture example with four threads), the processor consists of an instruction cache, an instruction TLB, a fetch unit (containing a branch predictor, program counters and a thread selector), a fetch queue, a decode unit, a register file, a scoreboard array, an instruction issue unit (containing an instruction issue queue), functional units (including no floating-point unit), a data cache, and a data TLB. Most resources in this processor are dynamically shared among threads with the exception of the few resources, as described below.

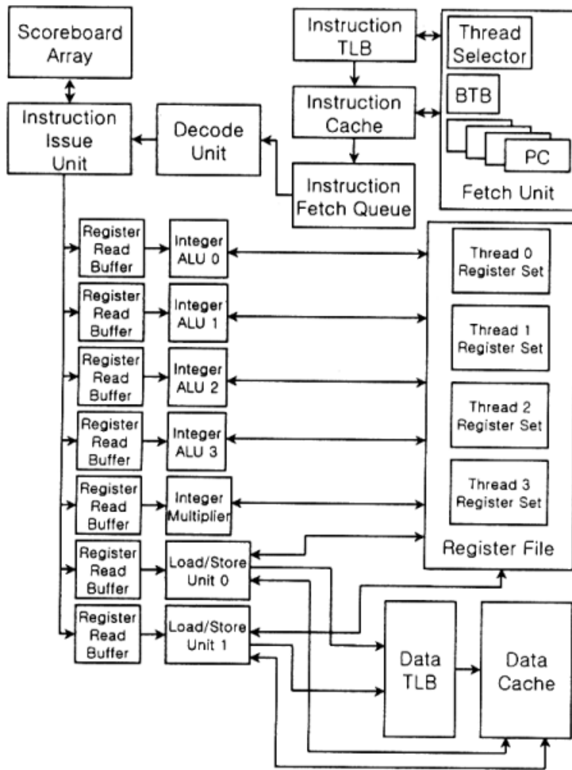

Fig. 1. Architecture example with four threads.

The fetch unit fetches instructions from multiple threads each cycle. If the fetch bandwidth is N instructions wide and the fetch unit partitions the bandwidth among M threads each cycle, the fetch unit fetches N/M instructions per thread from M threads selected by the thread selector every cycle. The thread selector picks up threads by a specific fetch priority algorithm. In this scheme, the instruction cache is nonblocking, since, during misses for some threads, the processor continues fetching instructions from other threads. The instruction cache consists of M banks and M ports with the purpose of fetching simultaneously instructions from M threads. In addition, the number of program counters (PCs) for fetching instructions must equal that of simultaneously supported threads (T) (one PC per thread). Figure 2 shows the instruction fetch structure.

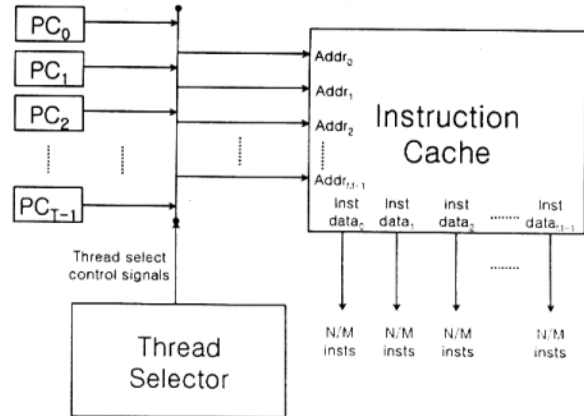

Fig. 2. Instruction fetch structure.

Fetched instructions are stored in the tail-pointed locations of the fetch queue, as shown in Fig. 3, in which the fetch queue has K entries. The decode unit decodes in order instructions from head-pointed locations of the fetch queue. During decoding, the decode unit determines what kind of functional unit is to execute the decoded instruction. At the same time, the decode unit identifies source registers, destination registers, immediate data and operation of each instruction. Finally, these in-order decoded instructions are stored in the instruction issue queue.

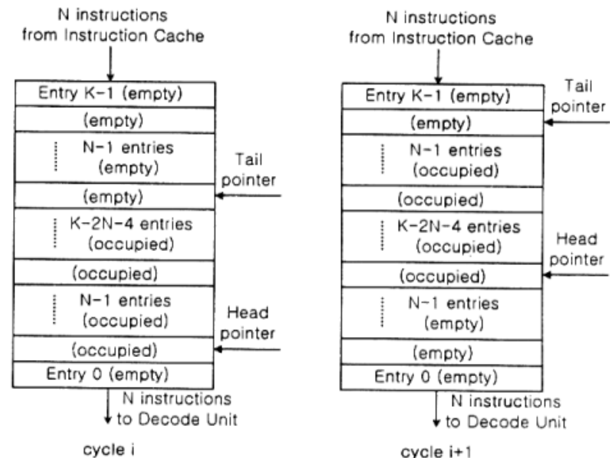

Fig. 3. Instruction fetch queue.

By checking the scoreboard array and the read stages of the functional units, the issue unit examines the oldest instructions of each thread in the instruction issue queue for their data dependencies and resource conflicts. Then, Instructions without data dependencies or resource conflicts are issued to their assigned functional units. Issued instructions register themselves to scoreboard entries corresponding to their destination registers. The issue order

of instructions from different threads may not be the same with the fetch order. Moreover, per-thread instruction flush is supported. These two factors cause empty (invalid) entries to occur between the head-pointed entry and tail-pointed entry. Therefore, compressing the instruction issue queue is necessary, as shown in Fig. 4, in which the instruction issue queue has L entries.

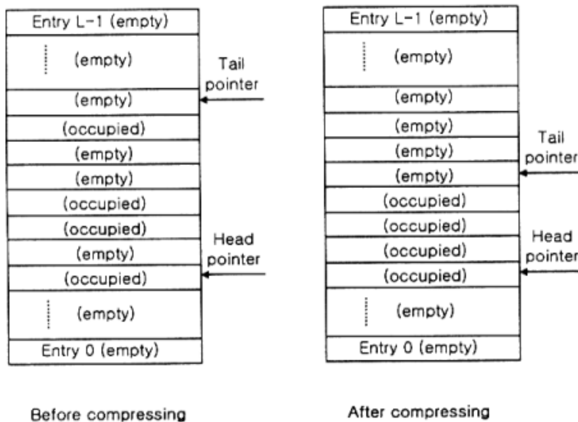

Fig. 4. Instruction issue queue compressing

In the read stage, issued instructions obtain their operands by result forwarding or read the operands from a register file, which contains the register sets for all the threads. Then, they are executed in functional units. Results of instruction executions are stored in result buffers, and forwarded to instructions in the read stage (if necessary for reducing latencies). Finally, the results are written back to the register file. Writes to the register file are performed in order, for each thread.

In this architecture, recovery for branch misprediction and exception is very simple thanks to its in-order completion feature, and is performed only by flushing the instructions that follow the mispredicted branch or exception instruction from the corresponding thread.

2.2 Pipeline Structure

The instruction pipeline of this processor is summarized as below.

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

2.3 Changes from Superscalar Architecture

Throughout the pipeline stages, each instruction is accompanied by its thread identifier. In addition, the following changes are made to support multiple threads simultaneously.

- The instruction cache that consists of several banks and is nonblocking.
- Multiple program counters and some mechanism by which the fetch unit selects threads.
- A thread id with each branch target buffer entry and a separate branch history buffer for each thread for predicting branches with higher accuracy.
- Per-thread instruction dependency checking.
- the instruction issue queue compressing.
- 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 above changes except the register file modification are trivial and the improved performance of our architecture that the simulation results show justifies these changes.

3. Simulation Results

In order to estimate our architecture, we implement a cycle-based, execution-driven simulator (based on the ARM architecture version 5) written in C language. Simulations with this simulator use independent programs (from the SPEC2000 benchmark suite) as threads and assume that there is no bank conflict in instruction and data caches. We use the cycle-based speed-up ratio as a performance criterion. Simulation results for various processor configurations are obtained by varying configuration parameters (e.g., fetch and issue bandwidth and the number of supported threads). The simulation results show that our processors with four and eight threads improve performance by two or more times over the conventional superscalar processor with comparable resources and issue policy. The simulation results for various configurations are

shown in Fig. 5 and Fig. 6, in which fetch thread selection uses round-robin algorithm, and issue thread selection is based on the number of instructions in the functional units for each thread.

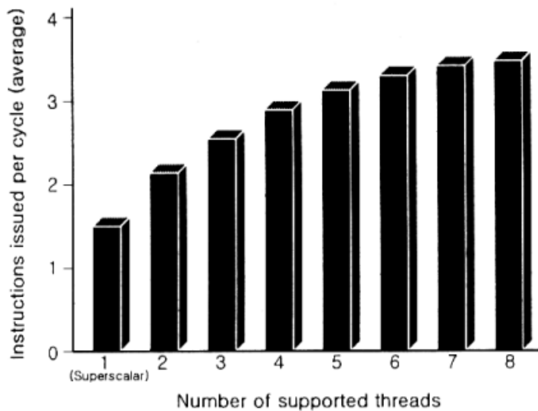

Fig. 5. Performance variation of four-issue processors

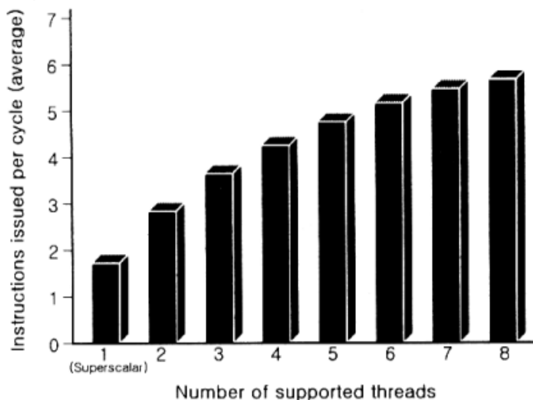

Fig. 6. Performance variation of eight-issue processors

4. Conclusions

In this paper, we have proposed a simultaneous multithreading architecture that issues and completes instructions in order for each thread. Our in-order issue and completion architecture reduces design complexity and hardware cost. 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 two or more times over the conventional superscalar processor with comparable resources and issue and completion policies.

Acknowledgements

The work reported in this paper has been 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.

References

[1] D.M. Tullsen, S.J. Eggers, and H.M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," Proc. 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, pp.392–403, June 1995.

[2] Keith Deifendorff, "Compaq Chooses SMT for Alpha," Microprocessor Report, Microdesign Resources, pp.1–11, Nov. 15 1999.

[3] 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 Annual International Symposium on Computer Architecture, Philadelphia, Pennsylvania, pp.191–202, May 1996.

[4] 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 Annual International Symposium on Computer Architecture, Queensland, Australia, pp.136–145, May 1992.