Dynamic Thread Extension for Speculative Multithreading Architectures

Niko Demus Barli,[†] Daisuke Tashiro,[††] Shuichi Sakai[†] and Hidehiko Tanaka[†]

To reduce the effect of thread overheads when executing small threads in speculative multithreading architectures, we propose a mechanism called Dynamic Thread Extension. This mechanism allows the hardware to dynamically combine two or more consecutive threads and treat them as a single thread. By having the compiler to estimate the size of threads, this mechanism can be implemented with minimal hardware support. Simulation results show that more an average of 13% performance improvement can be achieved.

1. Introduction

Speculative Multithreading has been employed in a number of microprocessor architectures to accelerate the execution of single threaded applications [1–7]. These architectures partition a single threaded program into threads either statically or dynamically. These threads are not necessarily independent, but may include some sort of control or data dependencies. It has been shown that considerable performance improvement can be achieved by executing these threads speculatively.

There are many challenges in designing a well-balanced speculative multithreading architecture. This paper is intended to deal with one of the design problems: how to reduce the effect of thread overheads in speculative multithreaded execution. We first verify the fact that there are many small threads created during a speculative multithreaded execution, thus introducing large overhead effect. It will be shown that the performance degradation due to this overhead effect may not be easily solved using software approaches alone. Consequently, we choose to combine both hardware and software approaches and propose an effective mechanism called Dynamic Thread Extension.

Dynamic Thread Extension dynamically combines two or more statically defined threads to form a larger thread, thus reducing the number of small threads executed. Compiler is responsible to estimate the size of these threads. This information is used by the hardware to decide whether it should *extend* the execution of a thread or not. By shifting some functionalities to the compiler, we show that Dynamic Thread Extension mechanism can be implemented with simple hardware while still achieving its purpose to reduce the effect of thread overheads.

The rest of this paper is organized as follows. Section 2 describes the baseline architecture and thread partitioning method used in this work. Section 3 investigates the effect of thread overheads and discusses a number of alternatives to reduce this effect. Section 4 describes the idea and implementation of Dynamic Thread Extension. Section 5 shows the evaluation results. Finally, section 6 concludes this paper.

2. Methodology

2.1 Baseline Architecture

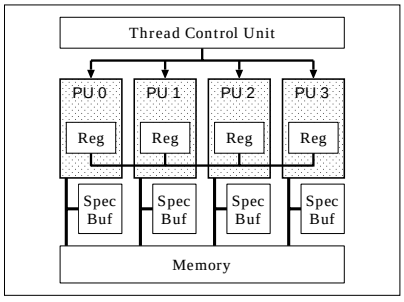

Fig. 1 Baseline Architecture

For the rest of this paper, we assume a baseline processor shown in figure 1. The processor is a Chip Multiprocessor (CMP), consisting of a thread control unit, four processing units, and a memory system. Thread control unit is responsible for resolving inter-thread control dependencies and scheduling threads into PUs. We assume a perfect next-thread prediction, i.e. the thread control unit always predicts successor threads correctly.

† Graduate School of Information Science and Technology, The University of Tokyo
†† Graduate School of Engineering, The University of Tokyo

Table 1 Baseline Architecture Parameters

No. of PUs	4 Processing Units
PU parameters	6-stage out-of-order superscalar 4 functional units 2 load/store units 4-instruction fetch width 64-entry instruction window 32-entry speculative buffer
Inst. Latency	2 cycles for Load/Store 1 cycle for other instruction
Delays	1 cycle thread start/stop ovh. 1 cycle communication delay 1 cycle restart ovh
Idealized conditions	Perfect memory Perfect next thread prediction

Each processing unit (PU) in this architecture is a 6-stage out-of-order superscalar core. Two-cycle execution latency is assumed for Load/Store instructions, whereas one-cycle execution latency is assumed for all other instructions. Table 1 summarizes the other parameters used in our baseline architecture.

To support speculative multithreaded execution, a register communication mechanism is assumed. Compiler statically analyzes a program for inter-thread register dependencies and inserts communication primitives into the program. Load instructions are speculatively executed. A mechanism to detect memory dependency violation is assumed. In case a violation occurs, the violating threads are squashed and restarted. Finally, each PU has a 32-entry speculative buffer to temporarily hold data speculatively written to memory.

2.2 Thread Partitioning Method

A program is statically analyzed and partitioned into threads. We define *thread* as a connected subgraph of a control flow graph with exactly one entry point. Overlapped regions shared by two or more threads may not exist. A thread partitioning algorithm proposed in [12] is used to put thread boundaries in function invocations and innermost loop iterations. For the remaining parts of the program, thread boundaries are put at places so that the resulting threads have a maximum size.

For clarity, it should be noted that when used in different contexts, the term *thread* may have different semantics. For the rest of paper, we will use the term *static thread* to refer to a portion of control flow graph defined as thread at compile time, and a plain *thread* to refer to a stream of instructions from a static thread actually executed by a PU.

3. Thread Overheads

3.1 Reducing Thread Overheads Effect

Many efforts have been done to reduce execution penalties originating from data misspeculation, control misspeculation, and synchronization/communication latency [8–11]. However, little attention has been paid to performance degradation caused by thread overheads. This problem arises when there are many small threads created during the execution. This paper focuses on this problem and offers a solution to reduce the effect of these overheads.

In general, thread overheads can be categorized into:

- **thread start overhead**: time required to schedule a thread into a processing unit
- **pipeline fill overhead**: time required for a processing unit to fill its pipeline before it can execute any instructions in the thread
- **pipeline drain overhead**: time required for a processing unit to retire remaining instructions left in the pipeline after it finished executing a thread
- **thread stop overhead**: time required to stop the execution of a thread and commit its speculative state to architectural state

For our baseline processor, the thread start and stop overhead are set to one cycle each. The sum of pipeline fill and drain overhead is approximately equal to the number of pipeline stages, which is six in our baseline processor. Thus, in our case, the total overhead is approximately eight cycles.

Since the trend in microprocessor design is toward deeper pipeline and more complex design, it is very unlikely that these overheads can be reduced. Thus, rather than trying reduce the overheads themselves, we should find a way to reduce their effect to processor performance. There are a number of alternatives available:

- Use simpler narrow-issue cores as PUs and increase the number of the PUs. This approach has some drawbacks. First, simpler narrow-issue cores will give inferior performance when used in traditional execution (single threaded execution). Second, the hardware cost and complexity for managing speculative multithreaded execution tend to increase when more PUs added.
- Introduce a mechanism to let a processing unit start fetching another thread as soon as there is no more instruction to be fetched from the current thread. This approach can

effectively hide the effect of thread overheads by overlapping the execution process of the two threads. However, it may introduce complexity and considerably increase the requirements of hardware support.

- Form larger threads so that the proportion of overhead to the actual execution cycles decreases. However, we may not freely create large threads since large threads generally impose strong inter-thread data dependencies.

Although the second approach also looks promising, we choose to employ the third approach. As it will be shown later, this approach can be implemented with simple hardware support while still giving substantial improvement in performance.

3.2 Thread Size and Performance

We conducted a preliminary evaluation to give a more concrete understanding on the relation between thread size and the performance achieved by speculative multithreaded execution. It is also intended to verify the trade-off between decreasing overhead effect and increasing data dependency when the threads are made larger.

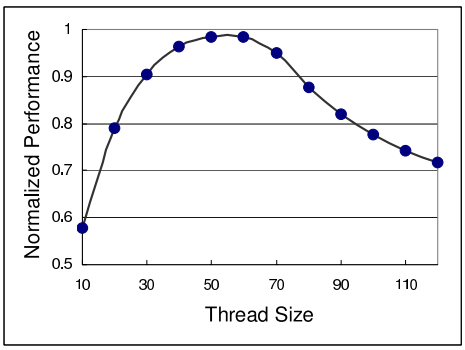

Fig. 2 Normalized performance with fixed thread size

Figure 2 shows normalized performance when a hypothetical thread partition algorithm which can create threads with a fixed size is assumed. Eight applications from Spec95int are used in this evaluation. As the thread size was varied from 10 to 50 instructions, the performance showed a significant improvement due to the decreasing thread overhead effect. However, when the thread size was further increased, the performance started to degrade. At this point, the limitation imposed by inter-thread data dependency governed the performance achieved by speculative multithreaded execution.

Although the absolute thread size values shown in this evaluation do not necessarily have important meanings, the evaluation results imply that size of threads executed should be adjusted to a range that compromises the trade-off of thread overheads and inter-thread data dependency. Dynamic Thread Extension, described in the following section, offers the flexibility to fulfill this requirement.

4. Dynamic Thread Extension

4.1 Basic Idea

The idea behind DTE is to combine two or more threads to form a larger thread whose size is within a range that compromises the trade-off of thread overheads and inter-thread data dependency. From preliminary evaluation result shown in figure 2, this range is approximately between 30-60 instructions for our baseline processor.

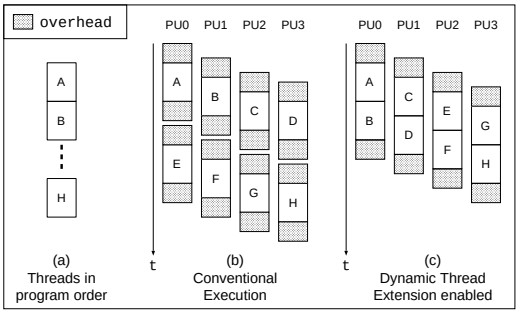

Fig. 3 Basic idea of Dynamic Thread Extension

Figure 3 illustrates the idea of DTE. Suppose we are executing threads A to H in a sequence shown in figure(a). Conventional speculative multithreading architecture schedules thread A to PU0, B to PU1, and so on, as shown in figure(b). If the size of these threads is small, thread overheads will take a significant portion of execution time. In contrast, DTE combines consecutive threads, thread A and thread B, thread C and D, and so on, if it founds the threads are small in size. The combined threads are then executed as single chunks in each PU, as shown in figure(c). In this way, the portion of overheads to execution time can be reduced and the overall performance of speculative multithreaded executed can be improved.

4.2 Why Dynamic Thread Extension ?

Figure 4 shows the distribution of number of instructions, categorized by the size of thread in which the instructions were executed. The partitioning method employed in this data is as described in 2.2. According to this figure, 56% of

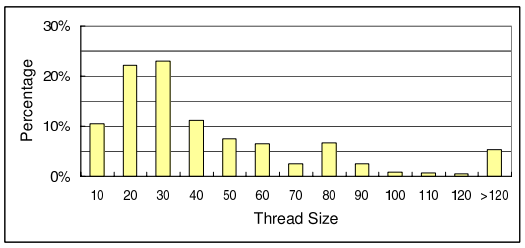

Fig. 4 Distribution of number of instructions executed within a thread whose size between $n-(n+10)$ instructions, where n is varied between $0-110$

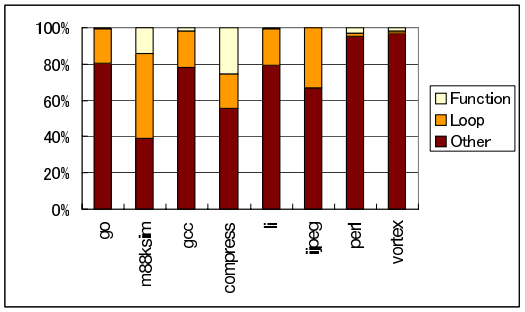

Fig. 5 Distribution of number of instructions, executed in threads whose size is less than 30 instructions

the executed instructions are belong to threads whose size is less than 30 instructions. It can be imagined that since the threads are very small, the performance suffers severely from the effect of thread overheads. Thus, a mechanism such as DTE is required to improve this situation.

A question of, whether conventional compiler techniques such as function inlining and loop unrolling are sufficient to do the task of creating larger threads, may arise. Function inlining reduces the number of small threads comprising functions as a whole, whereas loop unrolling is useful to enlarge the threads created from loop iterations. Our investigations however suggested that these techniques may not be sufficient. Figure 5 shows the distribution of number of instructions, executed in threads whose size is less than 30 instructions. These threads are classified into:

(1) function threads: threads that contains the whole function
(2) loop iteration threads: threads that are created from innermost loop iterations (one iteration per thread)
(3) other type of threads: threads that are not classified either to (1) or (2), generated mostly due to the restriction that thread boundaries should always be put at function calls

Function inlining and loop unrolling may help reduce the number of small threads of type (1) and (2). However, since for most applications these types of threads occupy only small portion of the total threads, we cannot expect adequate performance improvement by employing these techniques alone. Thus, a more universal approach is required. Since DTE is theoretically applicable to all types of threads, we may expect more performance improvement from it.

4.3 Implementation

DTE may be implemented completely in the hardware. However, since it is preferable to keep the hardware simple, we moved some functionalities into the software. The implementation is described as follows.

Compiler support

For each statically defined thread the compiler estimates its size (the number of instructions the thread will contain when executed). We employed a simple estimation method that takes the average size of all possible paths between the entry point and exit points of the thread.

After estimating the size, the compiler decide whether the thread should be marked *extendable* or *not-extendable*. If the estimated size is less than or equal to a preset threshold, the thread is marked extendable. Otherwise, it is marked not-extendable. We used two types of thread start instruction, *thrstart_ext* and *thrstart_noext*, to distinguish extendable from not-extendable threads.

Hardware support

Hardware to support DTE is implemented in each PU of the CMP. The hardware is responsible to decide whether the execution of an extendable thread should be extended to the successor thread, or be left unextended. We prepare a counter to count how many dynamic instructions has been fetched (not including instructions from mispredicted control path or violating execution). We also prepare an *extended flag* bit that when asserted indicates that the execution of a thread will be extended.

Figure 6 shows the state diagram of DTE hardware. The state *Normal* corresponds to the state when the *extended flag* is cleared, whereas the state *Extended* corresponds to the state when *extended flag* is asserted. The instruction counter is assumed to be reset to zero before an execution starts.

When the execution of a thread starts and the first instruction in the thread is a *thrstart_noext* then the state changes to *Normal*. In this case,

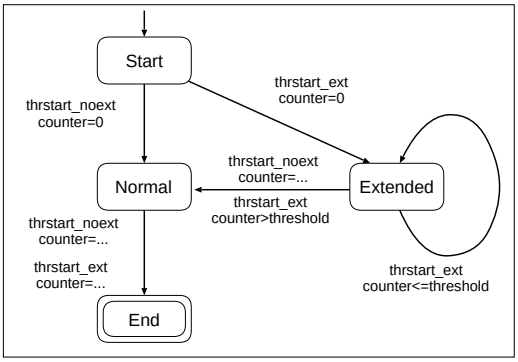

Fig. 6 State diagram of Dynamic Thread Extension hardware

the execution will terminate the next time it reaches any thread start instruction, either a *thrstart_noext* or a *thrstart_ext*. The value of instruction counter is ignored in both cases.

When the execution of a thread starts and the first instruction in the thread is a *thrstart_ext*, then the state changes to *Extended*. In this case, the execution of the thread will be extended to include at least the next one thread. It should be noted that in this state, all register communication instructions will be ignored since the register values may be redefined when the execution is extended.

When in *Extended* state, there are three possible state transitions of DTE hardware:

- If the next thread start instruction reached is a *thrstart_noext*, then the state changes to *Normal* and the execution will terminate the next time it reaches any thread start instruction.
- If the next thread start instruction reached is a *thrstart_ext* and the value of instruction counter is greater than a preset threshold, then the state changes to *Normal* and the execution will terminate the next time it reaches any thread start instruction.
- If the next thread start instruction reached is a *thrstart_ext* and the value of instruction counter is less than or equal to the preset threshold, then the state remains unchanged. The execution will be further extended.

As can be imagined from the above explanation, DTE can be implemented using a very simple hardware. Moreover, it requires no inter-PU communication. Thus, it is very unlikely that it becomes the bottleneck of the processor's critical path.

Assuming that the compiler can accurately estimate the thread size and the threshold value used by the compiler is less than or equal to the threshold value used by the hardware, DTE guarantees that the size of chunk of instructions the processor executed as a single thread is more than the hardware threshold. However, since the estimation cannot be 100% accurate, we could expect that there will exist a number of threads whose size is less than or equal to the hardware threshold.

5. Evaluation

We conducted simulations to evaluate the benefit of DTE in speculative multithreaded execution. Considering the result of preliminary evaluation already shown in figure 2, we set the compiler threshold to 20 instruction, and the hardware threshold to 30 instruction. We expected that larger portion of programs will be executed in threads whose size is larger than 30 instructions, thus reducing the effect of thread overheads and increasing the achieved performance.

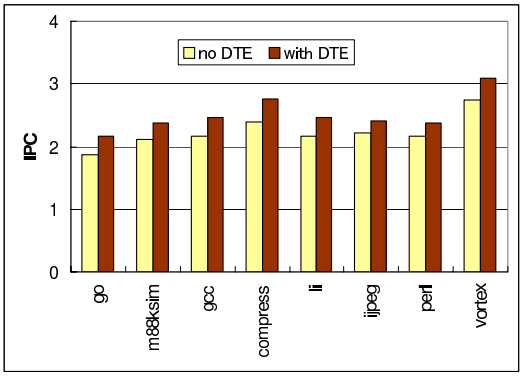

Fig. 7 Performance improvement by Dynamic Thread Extension

Figure 7 shows the results of simulations. When DTE is employed, all applications showed performance improvements, ranging from 8.5% to 16%, or 13% in average. Figure 8 shows the distribution of number of instructions executed within a thread whose size is within the specified range, when DTE is disabled and enabled. It becomes clear that DTE succeeded to reduce the fraction of program executed in small threads, thus, shifting the distribution graph to the right. The fraction of program executed in threads whose size is less than 30 instruction, for example, is reduced from 56% to 15%.

Our evaluation results shown above, verifies that DTE is an effective mechanism for reducing the number of small threads, thus reducing

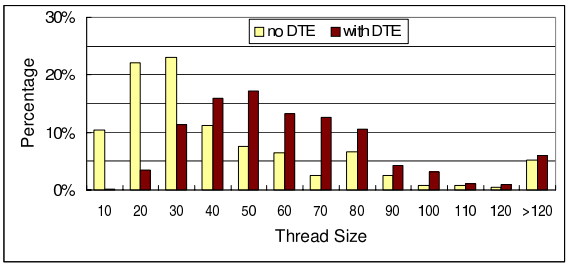

Fig. 8 Distribution of number of instructions executed within a thread whose size between $n-(n+10)$ instructions, where n is varied between $0-110$, with and without using DTE

the effect of thread overheads. It should also be noted that the potential of performance improvement by DTE is actually larger than what was shown above. There is a limitation in our current register communication scheme, i.e. a register value defined in an *Extended* state, may be sent to the consumer PU only after the execution enters *Normal* state, even though the register is never redefined any time later. In case an ideal register communication mechanism is assumed, DTE gives performance improvements between 15% to 37%.

6. Concluding Remarks

This paper investigated thread overheads problem, and verified its impact on the performance of speculative multithreaded execution. We investigated a number of possible solutions to the problem, and proposed a mechanism called Dynamic Thread Extension (DTE). By moving some functionalities to the compiler, we showed that DTE can be implemented using a simple hardware support. Despite the simplicity, our evalution results showed that DTE helps the processor achieving a performance improvement of 13% in average.

However, there are still a number of refinements needed to further improve the effectiveness of DTE. We plan to investigate the following issues:

- Register communication mechanism that can more effectively exploit the potential of DTE.
- Incorporating data dependence information for the compiler to decide whether a static thread should be marked extendable or not.
- Hardware refinement, for example rather than using a counter of number of fetched instructions, the hardware may should have to use a counter of execution cycles.
- The interaction between DTE and next-thread prediction mechanism.

Acknowledgement

This work is partially supported by Grant-in-Aid for Scientific Research, Ministry of Education, Culture, Sports, Science and Technology, Japan, Basic Research No.(B)(2)50291290.

References

[1] G.S. Sohi, S.E. Breach, and T. N. Vijaykumar, *Multiscalar Processors*, Proc. 22nd ISCA, pp. 414-425, 1995

[2] L. Hammond, M. Willey, and K. Olukotun, *Data Speculation Support for a Chip Multiprocessor*, Proc. 8th ASPLOS, pp. 58-69, San Jose CA, 1998

[3] 小林 良太郎, 岩田 充晃, 安藤 秀樹, 島田 俊夫, 非数値計算プログラムのスレッド間命令レベル並列を利用するプロセッサ・アーキテクチャ*SKY*, Proc. JSPP'98, pp.87-94, June 1998

[4] 鳥居 淳, 近藤 真己, 本村 真人, 西 直樹, 小長谷 明彦, *On Chip Multiprocessor* 指向 制御並列アーキテクチャ*MUSCAT* の提案, Proc. JSPP'97, pp.229-236, 1997

[5] V. Krishnan, J. Torellas, *A Chip-Multiprocessor Architecture with Speculative Multithreading*, IEEE Transactions on Computers, Vol. 48, No. 9, Sept 1999

[6] A. Gonzalez and P. Marcuello, *Speculative Multithreaded Processors*, Proc. 12th International Conference on Supercomputing, July 1998

[7] H. Akkary, M.A. Driscoll, *A Dynamic Multithreading Architecture*, Proc. 31st MICRO, Nov-Dec 1998

[8] T.N. Vijaykumar and G.S. Sohi, *Task Selection for a Multiscalar Processor*, Proc. 31st MICRO, Nov-Dec 1998

[9] 岩田 充晃, 小林 良太郎, 安藤 秀樹, 島田 俊夫, 制御等価を利用したスレッド分割技法, 97-ARC-128 pp.127-132, Mar 1998

[10] 堺 淳嗣, 鳥居 淳, 近藤 真己, 市川 成浩, 大俣 仁美, 西 直樹, 枝広 正人, 制御並列アーキテクチャ向け自動並列化コンパイラ手法, Proc. JSPP'98, pp.383-390, 1998

[11] R. Kobayashi, M. Iwata, Y. Ogawa, H. Ando, and T. Shimada, *An On-Chip Multiprocessor Architecture with a Non-Blocking Synchronization Mechanism*, Proc. 25th EUROMICRO, pp.432-440, Sept 1999.

[12] N.D. Barli, H. Mine, S. Sakai, and H. Tanaka, *A Thread Partitioning Algorithm using Structural Analysis*, ARC-2000-139 Vol. 2000, No. 24, pp. 37-42, Aug 2000

[13] C. Iwama, N.D. Barli, S. Sakai, and H. Tanaka, *Improving Conditional Branch Prediction on Speculative Multithreading Architectures*, Proc. JSPP 2001, pp. 165-172, Jun 2001