Vol.2010-ARC-190 No.27
2010/8/4

An Improvement of Switch-on-Future-Event Multithreading

NARUKI KURATA,[†1] RYOTA SHIOYA,[†1,†2] JUN NAKASHIMA,[†1]
MASAHIRO GOSHIMA[†1] and SHUICHI SAKAI [†1]

"Delinquent" instructions are a small number of static instructions that cause most branch prediction misses and cache misses in a program. One of the important features of those delinquent instructions is that most of them are executed in small loops. We have proposed a new scheme of multithreading called Switch-on-Future-Event Multithreading (SoFE-MT) that hides a latency of delinquent instructions by multithreading execution of a loop in a single program. The conventional SoFE-MT did not assume periodic memory cache misses or memory access order violation between threads which often occur in a loop. We propose a memory access prediction system and a memory confliction detection system to deal with such problems. Simulation results shows that our proposal achieves performance improvement by an average of 2.4% and a maximum of 15.3%.

1. Introduction

"*Delinquent*" instructions are a small number of static instructions that cause most branch prediction misses and cache misses in a program. One of the important features of those delinquent instructions is that most of them are executed in small loops[4]. In Section 2 , we point out many programs in SPEC CPU2006[1] have delinquent instructions in loops that consist of up to a few hundreds of instructions.

One of the typical topics for delinquent load instructions is prefetching.[6)5)] In most researches they prefetch data based on predicted access pattern, triggered by a cache miss. However, their techniques work only on a simple access pattern; otherwise, they need high cost predictors.

There is another effective technique to hide the latency of delinquent instruction; *multithreading*. There are two types of multithreading classified by degree of communication granularity among threads.

- **throughput oriented multithreading** The throughput oriented multithreading is to execute independent programs simultaneously in order to maximize total execution throughtput of the processor. For example, in *Switch-on-Event multithreading*, when a load instruction misses the cache, the processor switches the thread in order to hide the latency of memory access[2),3)]. However, the throughput oriented multithreading cannot improve execution throughput of each single program.
- **helper threading** The helper threading is to execute a *helper thread* earlier than the original thread in order to improve performance. The helper thread consists of a delinquent instruction and its dependent instructions only, so the helper thread can execute the delinquent instruction earlier than the main thread. The helper threading executes part of actual instructions so it can prefetch data from the memory more accurately than the prefetch system using prediction. Furthermore, the helper threading can also execute an actual branch instruction earlier than the main thread, so it can realize highly-accurate branch prediction using the result of the helper thread.

  However, if the distance between delinquent instructions is short, the helper thread cannot be executed sufficiently early. Furthermore, the helper thread becomes relatively large, so it disturbs the execution of the main thread. As most delinquent instructions are relatively in small loops, this is a big problem in such particular cases.

To hide a latency of branch prediction misses and cache misses occured by delinquent instructions, we have proposed *Switch-on-Future-Event multithreading(SoFE-MT)*[4]. In SoFE-MT the processor regard each iteration of a loop as a thread and executes them simultaneously with the structure of SMT. **Figure 1** shows hiding of the latency of a delinquent instruction. In this figure, the processor executes other iterations until it gets the result of branches. This implementation just switches threads and always executes the main context, while the helper threading executes extra instructions as a helper thread. Therefore, SoFE-MT is free from the problems which the helper threading suffers from.

SoFE-MT must guarantee that the result of the program is the same as the sequential execution. Consequently, the processor must resolve dependencies among threads. We have also proposed some techniques to deal with this problem in the conventional SoFE-MT model.

There are two problems in this implementation. First, the predictor cannot regard a load instruction which miss the cache periodically as a delinquent instruction. For

†1 Graduate School of Information Science and Technology, The University of Tokyo
†2 Research Fellow of the Japan Society for the Promotion of Science (DC2)

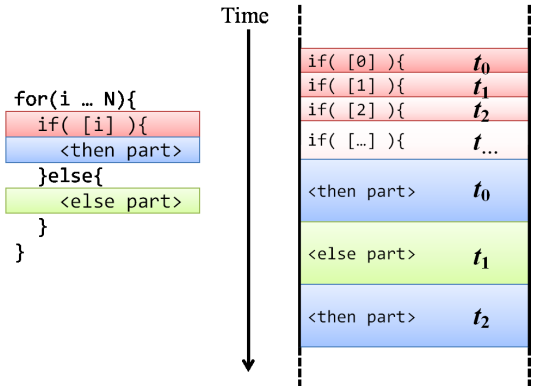

figure 1 Hiding Latency by Multithreading

example, if a load instruction accesses the memory sequentially, it misses the cache at first, and then hits for several times. When it goes off the cache line, the cache misses again. Such access pattern derives periodic cache misses. Although such case often occurs in loops, the processor cannot predict such instruction as a delinquent instruction, as it does not focus on dynamic features of delinquent instructions. Second, violation of memory access between threads occur frequently in some cases we state in Section 4.4 . We did not assume such cases so the performance degrade unexpectedly in the conventional implementation.

In this paper, we propose two new techniques for Switch-on-Future-Event multithreading to avoid those two problems. First is a new load hit/miss predictor in order to predict a load miss which occurs periodically. It can predict periodic cache misses including sequential cache accesses as stated above. Second, we implement a system of predicting a pair of load/store instructions that access the same address between different threads. The processor waits fetching the instruction of the following thread until the instruction of the precedent thread to avoid violation of memory access.

2. Delinquent Instructions and Loops

In recent processors, branch predictors are equipped in order to hide latencies of branch instructions. Similarly, they have the cache memory to reduce the latency of load instructions. However, some of the branch/load instructions cannot be solved by those systems. In this section, we take a look at such "*delinquent*" instructions.

2.1 Branch Instruction

When the processor fetches a branch instruction, it predicts the result of the instruction in order to keep fetching instructions and executing the program. For example, in a loop, most of the time a branch instruction shows the processor should execute the loop again. Thus, in many cases, directions of branch instructions have statistical features and the predictor can predict the right way.

However, some branch instrucions cannot be predicted in such a conventional way and they can be one of the causes of degradation of the processors. When a prediction misses, speculatively fetched instructions are flushed and the right instruction is going to be fetched. It takes some extra cycles to flush the mispredicted instructions so the efficiency of using the pipeline degrades.

One example of such instructions is a **virtual function**, as recent programming languages implement.

2.1.1 Example of Delinquent Instruction: Virtual Function

The virtual function is implemented in order to realize polymorphism in object oriented languages. The polymorphism reduces workload of programmers because a group of data which have similar characteristics can be handled with the same procedure by using polymorphism.

Figure 2 shows an example of a class that uses polymorphism. In this example, *Circle* and *Square* are derived classes of *Drawable* and *drawables* is a polymorphic list of these classes. Both *Circle* and *Square* override *Draw()* which is declared as a virtual function of *Drawable* class, so if we call *Draw()* as a function of *Drawable*, actually, it is called as a function of either *Circle* or *Square*. Which function is called depends on the past control flow, so it is difficult for conventional branch predictor to predict which way the branch instruction indicates.

2.2 Load Instruction

A load instruction transfers data from the main memory to a register in a few hundred cycles, so generally the processor has two or three layers of fast and small caches. The cache has a locality of space and time, so most of the time the cache hits. However, if a load instruction misses the cache, the following dependent instructions cannot be executed in many cycles. Especially, if a load instruction misses the cache every time, it can be a large factor of degradation of performance. To avoid this problem, there is a technique of prefetching data from the main memory in advance[5),6)].

```
class Drawable {
 public:
  virtual void Draw(void) = NULL;
};

class Circle : public Drawable {
 public:
  void Draw(void) { … };
};

class Square : public Drawable {
 public:
  void Draw(void) { … };
};

Drawable **drawables;
…
for (int i = 0; NULL != drawables[i]; i++) {
  drawables[i]->Draw();
}
```

figure 2 Example of Virtual Function.

In most researches of prefetching, they prefetch data based on a predicted access pattern, triggered by a cache miss. However, their techniques effectively work only on a simple access pattern: otherwise, they need extremely high cost predictors.

2.3 Relationship Between Delinquent Instructions and Loops

Delinquent instructions are a small number of static branch/load instructions that cause most of the branch prediction/cache misses. Consequently, one static instruction is executed many times. Generally, the program counter (PC) increases monotonically, so the same address of the instruction is executed twice or more means that a backward branch is executed once or more. In addition, an iteration structure with a backward branch is what we call a loop. Thus, we can say that a delinquent instruction is in loops.

Consequently, we focus on a relationship between sizes of loops and the number of delinquent instructions in the loops and run exploratory evaluation. We assume that the distance between the same PC of two different dynamic instructions as the size of the loop.

We used 29 programs from the SPEC CPU2006[1] benchmark with *ref* data sets. The parameters of the simulator are the same as we will mention at Section 6 .

Figure 3 and **figure 4** show the results of evaluation of branch prediction misses and L2 cache misses, respectively.

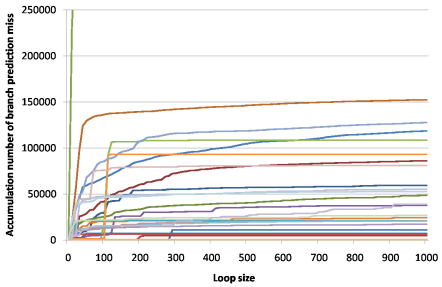

figure 3 Accumulation Number of Branch Prediction Misses at Every Loop Size

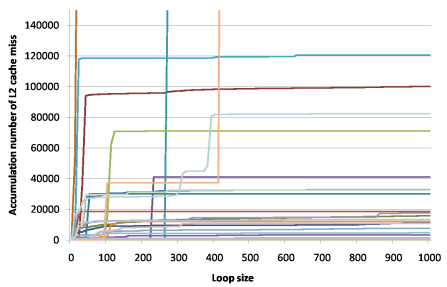

figure 4 Accumulation Number of L2 Cache Misses at Every Loop Size

In **figure 3**, in most of the programs, the number of branch prediction misses saturate within the loop which consists of 200 static instructions. figure 4 has the similar property as figure 3. In most programs, the number of branch prediction misses saturate within the loop which consists of 400 static instructions.

Thus, most of the delinquent instructions are executed relatively in small loops. In these loops, conventional techniques for delinquent instructions do not work effectively as we state in the next section.

3. Conventional Multhreading Techniques for Delinquent Instructions

The techniques of hiding the latency of delinquent instructions by multithreading is classified by degree of communication granularity between threads; the throughput oriented multithreading and the helper threading.

3.1 Throughput Oriented Multithreading

The throughput oriented multithreading processor basically executes independent programs simultaneously in order to maximize total throughput of the processor. The typical reseach of throughput oriented multithreading for delinquent instructions is *Switch-on-Event multithreading*[2),3)]. This technique is to switch a thread to another when a load instruction misses the cache in order to hide the latency of the cache miss. Furthermore, there are some reseaches for SMTs that speed up processors by optimizing not to fetch delinquent instructions[7),8)].

However, in these techniques, the processor switches the thread when an event such as a cache miss occurs. Therefore, the dependent instructions of the load instruction stuck

in the instruction window and degrades execution efficiency. Moreover, they cannot deal with delinquent branch instructions because when a branch instruction is executed and found that the branch predictor mispredicted, there is no chance to hide the latency of the misprediction. Furthermore, they improve total execution throughput of the processor, but they cannot improve execution throughput of each single program.

3.2 Helper Threading

The helper threading is to execute a helper thread earlier than the main thread. The helper thread consists of a delinquent instruction and instructions which the delinquent instruction depends on, so the helper thread can execute the delinquent instruction early and realize prefetching or accurate branch prediction. Among them, there are some reseaches that directly use the result of the helper thread and increase in performance[10]. The helper threading executes the actual instructions so it works even when a load instruction accesses the memory irregulary and simple prefetching mechanism does not work well.

However, the helper thread cannot lead the main thread when the distance between delinquent instructions is short, like small loops. Furthermore, in small loops, the helper thread is relatively large compared with the main thread. In such case, the helper thread prevents the main thread from executing its instructions and the performance degrades as a result. As noted above, most of the delinquent instructions are executed in relatively small loops, so the helper threading cannot improve the performance of the processor effectively.

3.3 Problem of Conventional Multithreading Techniques

The problem of conventional multithreading techniques can be summarized as follows.

As stated in Section 2.3 , delinquent instructions are mainly in small loops. To hide latency of the delinquent instructions effectively, we must consider this fact, otherwise the elaboration ends up with no effect. The throughput oriented multithreading has the problem that they improve total execution throughput of the processor, but they cannot improve execution throughput of each single program. Furthermore, in the Swith-on-Event multithreading, they cannot hide the latency of branch prediction misses. On the other hand, in the helper threading, the helper thread must lead the main thread sufficiently. However, if the distance between delinquent instructions is short like small loops, the helper thread cannot lead enough and the result of the helper thread misses the time for the main thread. To make matters worse, the helper threading consumes the resouces of the processor, so the main thread is interrupted to execute. This problem is unavoidable as long as the helper threading executes extra instructions.

Thus, these conventional is ineffective for most of the delinquent instructions which are in small loops.

4. Switch-on-Future-Event Multithreading

We have proposed the technique of hiding the latancy of delinquent instructions, called *Switch-on-Future-Event multithreading(SoFE-MT)* . This technique regards each iteration of a loop as a thread and executes them simultaneously with the SMT processor. As we mentioned above, most of the delinquent instructions are executed in small loops, so the processor cannot enjoy the benefits of the helper threading. On the other hand, SoFE-MT switches the thread from an iteration to another iteration, triggered by a fetch of a delinquent instruction. Therefore, SoFE-MT can hide the latency without suffering from the problem like them.

4.1 Behavior of Processor in Target Loop

To execute each iteration simultaneously with the same physical structure of the SMT processor, we propose some special instructions.

SoFE-MT starts when a *pstart* instruction is executed. **Figure 5** shows the allocation of threads on the number of n logical processors in the SMT processor. t_i in the figure shows the thread which corresponds to the ith iteration. At the beginning of the multithreading, 0th to $i-1$th thread are created on corresponding logical processors.

Each thread ends with *pend* instruction. When the pend is committed, the thread right after the number of logical thread is going to be created. In **figure 5**, for example, when the logical processor 0 has committed pend in the thread t_0, it creates the new thread of t_n on itself. An instruction *pexit* shows the end of the loop. When the pexit has committed, multithreading is finished and the processor executes next instructions with single threading.

Some threads in SoFE-MT use the result of precedent threads if there is a dependency among them. In that case, the processor must resolve those dependencies. We propose instructions *send* and *recv* to resolve this problem. Each logical processor is connected with unidirectional ring and communicates with neighbor processors. As we show in figure 5, the logical processor i sends data to the logical processor $(i+1)\%n$ ($\%$ indicates

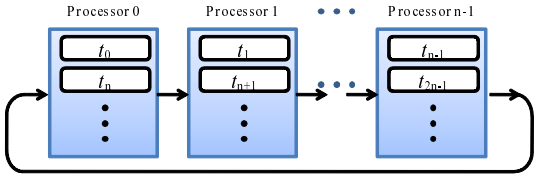

figure 5 Allocating Threads to n Logical Processors

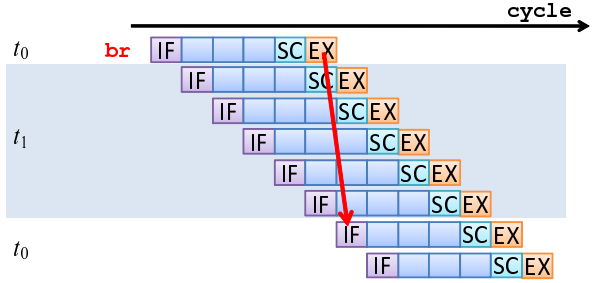

figure 6 Hiding Branch Instruction Latency by Multithreading

a modulus operator).

These special instructions are inserted statically by the compiler.

4.2 Hiding Latency of Delinquent Instructions

The processor predicts if a branch/load instruction is a delinquent instruction with saturation counters in the table we call *Delinquent Instruction Table(DIT)*. A counter is incremented when the branch prediction or the cache misses and decremented if it hits. Most of the instructions that does not miss the branch prediction or the cache never consumes the entry of the DIT. Therefore, the DIT can be small enough to work effectively.

The processor determines if the instruction is a delinquent instruction by the MSB of the counter when it fetches instructions.

For example, **figure 6** shows hiding of the latency. In this figure, the instructions of the pipeline chart that the background is not painted belong to thread t_0, while the background is painted belong to thread t_1. This figure shows a branch instruction br of t_0 is predicted as a delinquent instruction. In this situation, the processor fetches t_1 and keep executing the program. As the br has executed, the processor switch the thread again and fetches instructions of t_0. In this example, the processor does not use branch prediction to hide the latency of br. This is one example for a branch instruction, but the processor can do the same thing to load instructions. When it fetch a load instruction that seems to miss the cache, it switches the thread. This prevents instructions after the load instruction from being stacked in the processor.

4.3 Detail of Implementation

The Switch-on-Future-Event multithreading guarantees that the result of the execution is the same as the sequential execution. Consequently, in genaral, the thread of the following iteration cannot execute until the thread of the precedent iteration has finished executing all the instructions. In this situation, instructions of the following thread get stuck in the instruction window and degrade the performance of the processor. To make matters worse, these instruction might cause deadlock of the instructions. To avoid this problem, we have proposed the technique of controlling the optimal thread switching. Additionally, the following threads commit their instructions tentatively. In this case, a memory access order violation might occur. To deal with this problem, we have also implemented memory confict detection system.

4.3.1 Control of Thread Switching

If the processor fetches recv of the following thread and its dependent instructions before fetching send of the preceding thread, they stuck in the instruction window and results in degradation of performance. To avoid this problem, the processor fetches the most preceding thread basically.

To realize this, we add some *Wait Delinquent instruction Flag(WDF)* and *Iteration Number(IN)* to each logical processor. WDF shows that a delinquent instruction is fetched and not finished executing in the logical processor. On the other hand, IN shows an iteration count from the beginning of the loop. The processor decide which thread to fetch with WDFs and INs. It fetches the thread which WDF is not set and has the smallest IN among them. If all the WDFs of the threads is set, the processor fetches the most preceding thread.

4.3.2 Conflict Detection

To guarantee the results of execution, if a memory instruction raise access order violation between threads, the processor should revert the thread. We detect this violation with a system similar to **Transactional Memory**[11]. The deffierence between Transactional Memory and the proposal technique is that the threads have priority.

To detect memory confliction, we add *Speculative Read bits(SR bits)* and *Speculative Write bits(SW bits)* to each L1 cache line. The number of SR bits and SW bits are

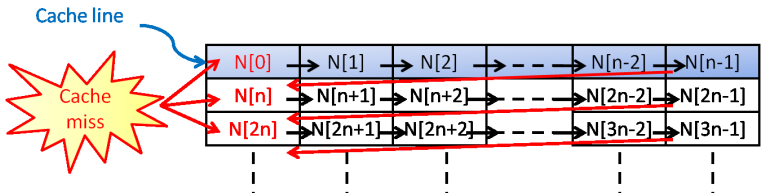

figure 7 Periodical Cache Miss on Array

the same as the number of logical processors. These bits indicate that a thread has loaded/stored data to the cache line, respectively. Each logical processor checks these bits when they access a cache line and if any bit is set, the processor flushes the following threads except only SR bits are set and the thread accesses the cache line to load the data. In addition, the SR bit and the SW bit of the most preceding thread are always unset because the memory access of the thread is always non-speculative.

To avoid occuring those violation so frequently , the processor adds the instruction just before the instruction which cause violation to DIT. The WDF of the thread that stops with this delinquent instruction is not unset until the preceding thread fetches pend.

4.4 Problem of Conventional SoFE-MT

There are two preblems in this model. First, the processor cannot regard a load instruction which miss the cache periodically as a delinquent instruction. For example, the processor often accesses an array sequentially in a loop. In such case, if the array has not been set on the cache, the cache line misses periodically(**figure 7**). However, in the conventional SoFE-MT, we cannot predict such an easy pattern. The DIT can predict if an instruction has a static characteristic, but it cannot predict when a load instruction has a dynamic characteristic of cache misses.

Second, memory confliction still occurs even we implement the technique mentioned in Section 4.3.2 . In that implementation, a pair of those memory instructions are put into the instruction window simultaneously and the instruction of the following thread may be executed earlier. This results in continual memory access violation and degrades execution efficiency.

5. Improvement of Switch-on-Future-Event Multithreading

The SoFE-MT system is better than other conventional multithreading techniques as we stated above. However, there are some problems in the prediction of cache misses and detecting memory access order violation. We propose *Stride Hit Miss Predictor* and *Memory Confliction Detection System* to solve these problems.

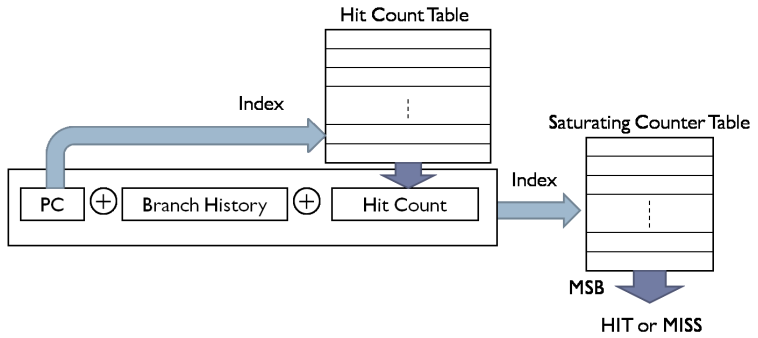

figure 8 Stride Hit Miss Prediction

5.1 Stride Hit Miss Prediction

Prediction of load hit/miss is important for some processor architectures. For example, the Alpha 21264 microprocessor architecture adopts a load hit/miss predictor in order to handle issue timing of instructions which are dependent on a load instruction[12]. The predictor on Alpha 21264 is the MSB of a 4-bit saturating counter which decrements by two when there is a load miss, otherwise it increments by one when there is a hit.

In the SoFE-MT, however, such predictor does not work effectively because it cannot predict periodical cache misses like figure 7. Consequently, we propose a new load hit/miss prediction system for the SoFE-MT showed in **figure 8**.

There are two tables in the system. The *Hit Count Table(HCT)* increments when a load instruction hits the cache, otherwise it resets to zero when it misses the cache. It is indexed by the PC of a load instruction, so it can count the number how many times each static instruction hits the cache consecutively. Each line of The HCT has a confidence counter and the instruction is predicted as a cache hit when the prediction is unreliable, because load instructions often hit the cache. A static instruction is added to the HCT only when it misses the cache, so the HCT can be small enough to get sufficient perfomance. On the other hand, *Saturating Counter Table(SCT)* increments when a load instruction misses the cache, otherwise it decrements when it hits the cache. The SCT is indexed by a convoluted number of the PC, the branch history and the hit count produced by the HCT. We convolute the branch history in order to deal with multiple loops. The

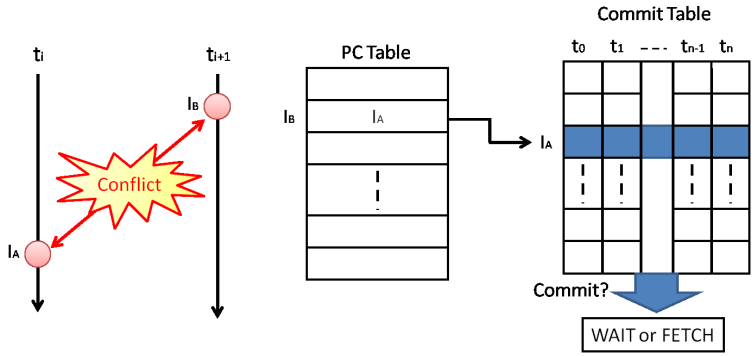

figure 9 Memory Confliction Detection System

MSB of the SCT is a prediction result whether a load misses or not. Thus, A load instruction can be predicted if it misses the cache periodically.

5.2 Memory Confliction Detection System

In the conventional SoFE-MT, if the preceding thread fetches pend the following thread starts fetching instructions even when the WDF is set by memory confliction. This implementation has possibilities to occur the memory access order violation again as we stated above. To avoid this, we proposed *Memory Confliction Detection System*. Memory Confliction Detection System consists of the PC table and the commit table. When access order violation occurs, the PC of the instruction in the preceding thread is put in an entry of the PC table, indexed by the instruction in the following thread. The commit table has an equal number of bits to the number of logical threads in each entry, which shows if the instruction of the corresponding thread has been committed.

Figure 9 shows the behavior of Memory Confliction Detection System. When an access violation occurs, the PC of the instruction in the preceding thread is registered in the PC table, indexed by the PC of the instruction in the following thread. The following thread always access to the PC table when it tries to fetch a memory access instruction. If a PC is registered in the corresponding entry, it accesses the commit table using the PC in the table. If the commit table shows that all the preceding threads have committed the corresponding instruction, the processor fetches the instruction. Otherwise, it waits for all the preceding threads have committed the instruction. The number of static load instruction that occurs access violation is small as the number of static instruction is small in loops, so the tables in this system can be small enough.

table 1 Simulation Configurations

パラメータ	値
ISA	Alpha 21164A
logical thread	4 way
fetch width	4 inst.
execution unit	int : 2, fp : 2, mem : 2.
instruction window	int : 32, fp : 16, mem : 16
register file	int : 256, fp : 256
branch prediction	8KB g-share
miss penalty	10 cycle
BTB	2K entry, 4-way
L1C	32KB, 4-way, 64B/line, 2 cycle
L2C	4MB, 8-way, 64B/line, 10 cycle
main memory	100 cycle

table 2 Evaluated Benchmarks

Benchmark Sets	Applications
SPECCPU 2006[1)]	perlbench, mcf, hmmer, h264ref, astar
MediaBench[13)]	adpcm_dec, adpcm_enc
EEMBC[14)]	dither
OOCSB A C++ benchmark[15)]	deltablue

table 3 Parameters of Switch-on-Future-Event

Name	Value
DIT	1KB, 4-way, 3 bits/count
HCT	160B, 2-way 8 bits/count 2 bits/conf
SCT	256B, 2 bits/count
PC Table	256B, 2-way
Commit Table	16B, 2-way

6. Evaluation

We evaluated using the following models on a cycle-accurate processor simulator *Onikiri2*[16)]. Parameters for the evaluation are showed in **table 1**.

Table 2 shows the benchmarks we used for evaluation. We modified the compiler gcc-4.3.3 to insert special instructions mentioned in 4.1 . We used the compile option “-O3” to compile benchmarks. We selected the benchmarks which delinquent instructions are obvious and insert the instructions to the corresponding loops using dedicated pragma for SoFE-MT.

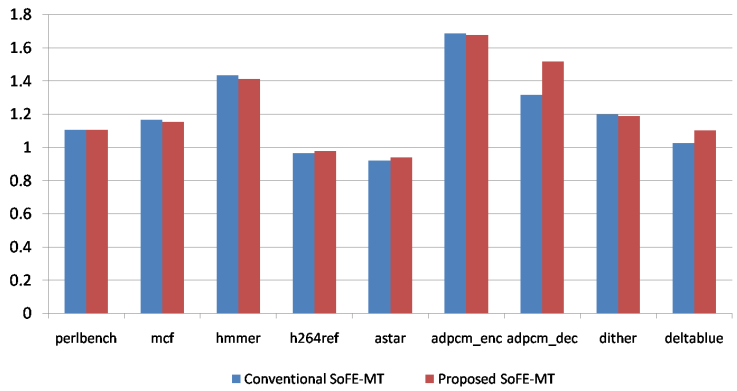

figure 10 Relative IPC of SoFE-MT to Single Thread Program

We skipped 1G instructions and evaluated the next 100M instructions except Mediabench benchmark. As the number of instructions is small, we executed all the instructions in Mediabench applications.

Table 3 shows the configuration parameters of SoFE-MT. Each entry of the DIT increments by two when the branch/cache hit prediction misses, while decrements by one when it hits. The instruction is regarded as a delinquent instruction when the entry of the instruction is not zero. On the other hand, each entry of the SCT increments by one when a load instruction misses the cache, otherwise it decrements by one when it hits the cache.

In the evaluation, we detect memory access order violation only when different threads access the same memory address other than the same cache line.

6.1 Results

Figure 10 shows the relative IPC of conventional SoFE-MT and the proposed SoFE-MT to the single thread programs. The proposed SoFE-MT model achieves performance improvement by an average of 23.1% and a maximum of 67.7% from the single thread model. Compared with the conventional SoFE-MT model, the proposed model achieves performance improvement by an average of 2.4% and a maximum of 15.3%.

Some applications show slight performance degradation by a maximum of 1.4% compared with the conventional SoFE-MT. This degradation of performance occurs because the threads wait too much by Memory Confliction Detection System system. In addition, the application astar still degrades performance by 6.0% compared with the single thread model. This is because a recv instruction and its dependent instructions wait for the send instruction of the preceding thread and stuck in the instruction window.

7. Conclusion

In this paper, we proposed two new systems to improve SoFE-MT which hide the latency of delinquent instructions. The Stride Hit Miss Prediction System enables to detect delinquent load instructions that occur periodical cache miss. On the other hand, the Memory Confliction Detection System suppresses continual memory access order violation between threads.

Our evaluation shows that the proposed model of SoFE-MT achieves performance improvement by an average of 2.0% and a maximum of 15.3% from the conventional SoFE-MT. However, some applications show slight performance degradation. This is because instructions of following threads stuck in the instruction window and degradate execution efficiency. Our plan for the future study is to solve such degradation.

Acknowledgement

This research was partially supported by Grant-in-Aid for Scientific Research No.20300015.

References

1) The Standard Performance Evaluation Corporation: *SPEC CPU2006 suite http://www.spec.org/cpu2006/*.
2) Farrens, M. and Pleszkun, A.: Strategies for achieving improved processor throughput, *ISCA*, pp.362–369 (1991).
3) McNairy, C. and Bhatia, R.: Montecito: a dual-core, dual-thread Itanium processor, *Micro, IEEE*, Vol.25, No.2, pp.10–20 (2005).
4) Shioya, R., Kurata, N., Nakashima, J., Goshima, M. and Sakai, S.: Switch-on-Future-Event Multithreading, *Symp. on Advanced Computing Systems & Infrastructures*, pp.157–165 (2010).
5) Palacharla, S. and Kessler, R.: Evaluating stream buffers as a secondary cache replacement, *ISCA*, pp.24–33 (1994).
6) Joseph, D. and Grunwald, D.: Prefetching using Markov predictors, *Computers, IEEE Transactions on*, Vol.48, No.2, pp.121–133 (1999).
7) Luo, K., Franklin, M., Mukherjee, S. and Sezne, A.: Boosting SMT performance by speculation control, *International Parallel and Distributed Processing Symposium*, pp.9 pp.– (2001).

8) Eyerman, S. and Eccklhout, L.: A Memory-Level Parallelism Aware Fetch Policy for SMT Processors, *HPCA*, pp.240–249 (2007).
9) Liao, S.S., Wang, P.H., Wang, H., Hoflehner, G., Lavery, D. and Shen, J.P.: Post-pass binary adaptation for software-based speculative precomputation, *Proceedings of Conference on Programming language design and implementation*, New York, NY, USA, ACM, pp.117–128 (2002).
10) Roth, A. and Sohi, G.: Speculative data-driven multithreading, *HPCA*, pp.37–48 (2001).
11) Herlihy, M. and Moss, J. E.B.: Transactional Memory: Architectural Support for Lock-Free Data Structures, *ISCA*, pp.289–300 (1993).
12) Mclellan, E. and Webb, D.: The Alpha 21264 Microprocessor Architecture, *ICCD*, p. 90 (1998).
13) Lee, C., Potkonjak, M. and Mangione-Smith, W.: MediaBench: a tool for evaluating and synthesizing multimedia and communications systems, *MICRO*, pp.330–335 (1997).
14) The Embedded Microprocessor Benchmark Consortium: *http://www.eembc.org/*.
15) OOCSB: Object-Oriented Compilers at UCSB: *A C++ Benchmark Suit http://www.cs.ucsb.edu/urs/oocsb/*.
16) Shioya, R., Goshima, M. and Sakai, S.: Design and Implementation of a Processor Simulator "Onikiri2", *Symp. on Advanced Computing Systems & Infrastructures*, pp.120–121 (2009).