Master's Thesis

Switch-on-Future-Event: A New Multithreaded Architecture for Single Programs

(Switch-on-Future-Event: 単一プログラムを高速化する新しいマルチスレッドアーキテクチャ)

Supervisor: Professor Shuichi Sakai

48-106410 Naruki Kurata

DEPARTMENT OF INFORMATION SCIENCE AND TECHNOLOGY,
THE UNIVERSITY OF TOKYO

February 8th, 2012

abstract

Delinquent instructions are a small number of static instructions that cause most branch prediction misses and cache misses in a program. These delinquent instructions are one of the main factors that degrade the performance of recent processors. One multithreading scheme that hides the latency of such delinquent instructions and speed up a single program is called Helper Threading. Helper Threading creates a helper thread which consists of a delinquent instruction and the instructions it depends on, and executes them earlier than the main thread to achieve accurate branch prediction or prefetching. However, we found an important feature of the delinquent instructions that most of them are executed in small loops. In such a small loop, Helper Threading cannot improve performance because it not only cannot do pre-execution sufficiently earlier than the main thread, but also prevents it from executing.

We propose a new scheme of multithreading called Switch-on-Future-Event Multithreading (SoF-MT). SoF-MT regards each iteration of a loop as a thread and executes them simultaneously. The processor switches a thread when it fetches a delinquent instruction to hide the latency of a miss that the instruction will cause in the future. This technique works well because delinquent instructions are in small loops and the processor can create a sufficient number of threads to switch. This scheme is free from the problems which Helper Threading suffers from. Simulation results show that our proposal achieves performance improvement by an average of 10.1% and a maximum of 38.7%, whereas Helper Threading provides only 13.2% speedup at a maximum.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Programs essentially contain fragments that cannot be parallelized. As Amdahl's law suggests, speed-up of a single core which executes sequential portion of programs will remain to be important even in the era of many-core processors. In fact, the major processor vendors are maintaining the widths of each core of mainstream multi-core processors.

"*Delinquent*" instructions are static instructions that cause most branch prediction misses and cache misses in a program. It is known that cache misses and branch prediction misses are caused by a small number of delinquent branch instructions. These delinquent instructions are one of the main factors that degrade performance of recent processors.

One of the typical topics for delinquent load instructions is *prefetching* [7][14]. In most researches they prefetch data based on a predicted access pattern, triggered by a cache miss. However, their methods work only on a simple access pattern; otherwise, they need high cost predictors.

There is another effective technique to hide latencies of delinquent instructions; *multithreading*. There are two types of multithreading, classified by the goals.

1. **Throughput-Oriented Multithreading** Throughput-Oriented Multithreading is to execute independent programs simultaneously to maximize the total execution throughput 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 [5, 12]. Besides, there are some methods for SMTs not to fetch a thread that is executing more delinquent instructions [4, 10]. These methods avoid fetching instructions that are going to be flushed or stall and consume resources of the processor.

   Switch-on-Event Multithreading switches a thread when an actual miss occurs. Therefore, it cannot hide the latency of a branch instruction. On the

other hand, SMT methods can hide latencies of both branch and load delinquent instructions. However, as they assume that they are executing independent programs, they do not improve the execution speed of each single program.

2. **Helper Threading** Helper Threading is to execute a *helper thread* earlier than the original thread to improve performance of a single program. The helper thread consists of a delinquent instruction and its dependent instructions, so the helper thread can execute the delinquent instruction earlier than the main thread. The Helper Threading executes part of the 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 predictions using the result of the helper thread.

One of the important features of those delinquent instructions is that most of them are executed in small loops. In Section 2, we point out many programs in SPEC CPU2006 [19] have delinquent instructions in loops that consist of up to a few hundreds of instructions.

There are many researches about Helper Threading, but it is not taken into account that most of the delinquent instructions exist in small loops. If the distance between delinquent instructions is short, as in a small loop, the helper thread cannot be executed sufficiently earlier than the main thread. Furthermore, as Helper Threading executes a helper thread that consists of extra instructions, the number of helper threads becomes larger, so they disturb the execution of the main thread. These helper threads in small loops not only cannot lead the main thread enough, and thus achieve nothing, but also result in degradation of performance.

To hide latencies of branch prediction misses and cache misses occurring from delinquent instructions in small loops, we propose *Switch-on-Future-Event multithreading (SoF-MT)*. In SoF-MT, the processor regards each iteration of a loop as a thread and executes them simultaneously with the structure of SMT. **Figure 1.1** shows hiding of the latency of a delinquent instruction. In this figure, the processor switches the thread to another when it fetches a delinquent branch instruction and fetches other iterations until the branch instruction is executed. When the instruction is executed and the branch destination is identified, the processor switches to the original thread. This implementation just switches threads and always executes the main context, while Helper Threading executes extra instructions as a helper thread. Therefore, SoF-MT is free from the problems which Helper Threading suffers from.

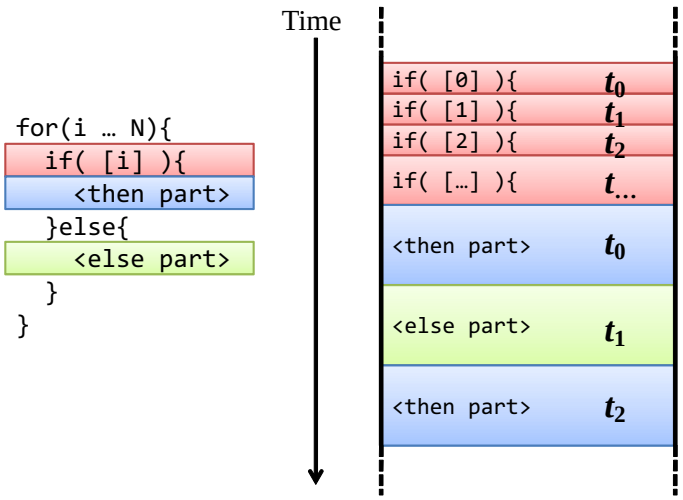

Figure 1.1: Hiding Latency by Multithreading

Throughput-Oriented Multithreading methods using SMTs do not take into account that there is a dependency between threads because they intend to execute independent programs. On the other hand, SoF-MT must guarantee that the result of a program is the same from the sequential execution. Consequently, in general, the thread of the following iteration cannot execute until the thread of the preceding iteration has finished executing all the instructions. In such a case, if the processor simply switches a thread, instructions of the following threads get stuck in the instruction window and degrade the performance of the processor. To make matters worse, these instructions might cause a deadlock of the instructions.

To avoid this, we also propose *optimal thread switching policy* and *speculative retirement*. We focus on the characteristics that delinquent instructions are a small number of static instructions and find these instructions with a few small tables to predict misses accurately. In addition, we allow following threads to retire without waiting for the preceding threads. To keep consistency of the program, we manage memory accesses with a small cache called *Transactional Data Cache*.

The rest of the paper is organized as follows. Section 2 describes that delinquent instructions are relatively often in small loops. After introducing conventional multithreading techniques for single programs in Section 3, Section 4 describes an overview of SoF-MT. Then, we take a look at the details of SoF-MT in Section 5 and Section 6 presents the evaluation results.

Chapter 2

A Factor of Degrading Performance in Executing A Single Program

2.1 Delinquent Instructions

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.

2.1.1 Delinquent Branches

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.

2.1.2 Delinquent Loads

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[14, 7].

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. Thus, these techniques are difficult to realize.

2.2 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. That is, one static instruction is executed many times. Generally, the program counter (PC) increases monotonically, so that the same address of the instruction is executed twice or more means that a backward branch is executed once or more. In addition, in most cases, an iteration structure with a backward branch is what we call a loop. Thus, we can say that a delinquent instruction is in a loop.

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

We used a cycle-accurate processor simulator developed in our laboratory and ran 29 programs from the SPEC CPU2006[19] benchmark with *ref* data sets for simulation. The parameters of the simulator are given in Section 6.

Figure 2.1 and **figure 2.2** show the cumulative number of branch prediction misses and L2 cache misses respectively, when changing the size of loops. For example, the point where the loop size is 400 and the cumulative number of misses is 10000 indicates that 10000 misses occur in loops which have 400 or fewer instructions in them.

In **figure 2.1**, in most of the programs, the number of branch prediction misses saturates within the loop which consists of 200 instructions. Figure 2.2 has the similar property as figure 2.1. In most programs, the number of branch prediction misses saturates 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.

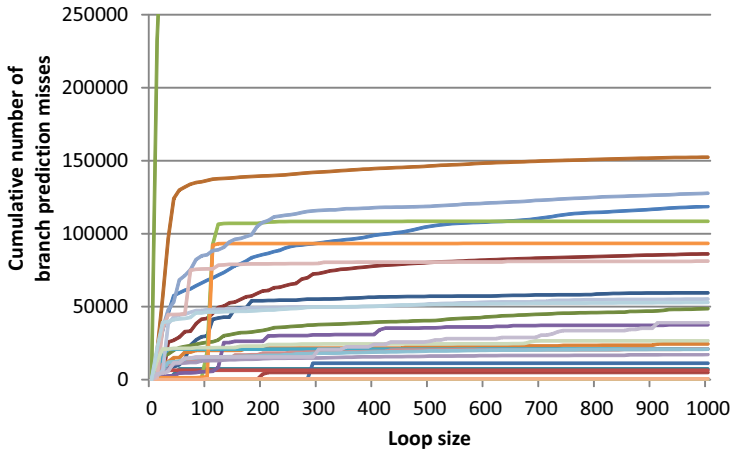

Figure 2.1: Cumulative Number of Branch Prediction Misses at Every Loop Size

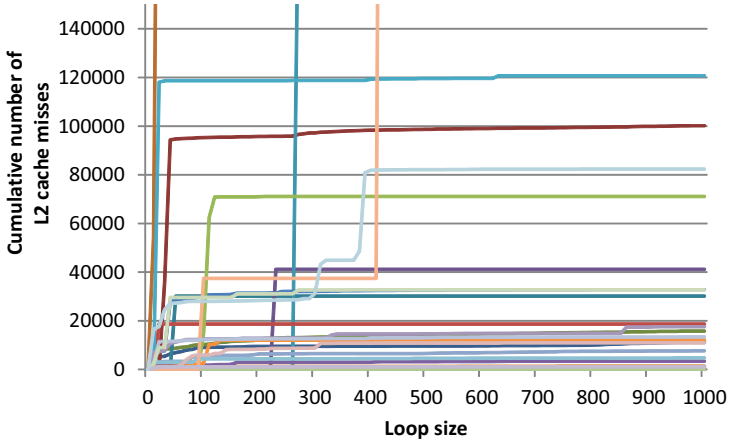

Figure 2.2: Cumulative Number of L2 Cache Misses at Every Loop Size

Chapter 3

Multhreaded Processor for Single Programs

There are some conventional multithreading techniques for single programs. In this section, we introduce *Speculative Multithreading* and *Helper Threading* to speed up single programs.

3.1 Speculative Multithreading

Speculative Multithreading (SpMT) is one of the well-studied methods to speed up single programs by extracting thread-level parallelism. In SpMT, the processor executes multiple threads that consist of parts (typically basic blocks) of a single program in parallel. Multiscalar[18], SKY[16], MUSCAT[20] and Pinot[13] are typical examples of SpMT.

Hardware A processor of SpMT consists of multiple suparscalar units, each of which is called a Processor Element (PE). A PE is connected to neighboring processors with a unidirectional ring and the PE can send data in registers to another PE.

Multithreading Model As we mentioned above, a thread for SpMT consists of a part of a single program and the processor executes them in parallel. A program is devided into threads by a compiler and the timing of forking a thread is also determined by the compiler.

Figure 3.1 shows how SpMT executes a program. Figure 3.1 (a) shows that a processor sequentially executes the program, while (b) shows that the program is executed by a SpMT processor. A PE forks the next thread in the neighboring PE when a fork instruction inserted by a compiler is executed.

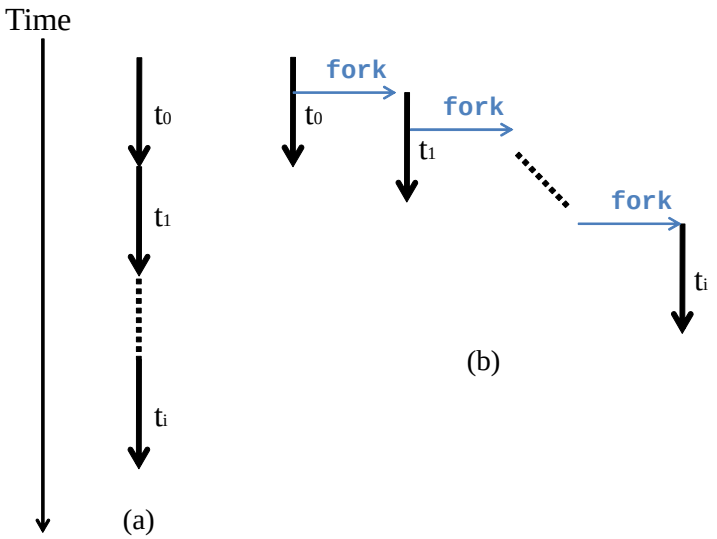

Figure 3.1: (a) Sequential Execution (b) Parallel Execution with Speculative Multi-threading

Delinquent Instructions Although SpMT extracts thread-level parallelism and speed up a single program, it does not hide latencies of delinquent instructions. The processor can hide latencies of delinquent loads when each delinquent instruction is executed in different PE as a whole, but the following instructions of a delinquent loads in the same PE should be stalled. Moreover, SpMT is completely ineffective for delinquent branches.

3.2 Helper Threading

The Helper Threading executes a helper thread earlier than the main thread in an SMT processor. The helper thread consists of a delinquent instruction and instructions on which the delinquent instruction depends, so the helper thread can execute the delinquent instruction early and realize prefetching or accurate branch prediction.

Figure 3.2 shows the pipeline chart of Helper Threading. In this figure, IF and EX indicate instruction fetching and execution. Pipelines with painted and not painted background belong to the helper thread and the main thread, respectively. In the figure, the branch instruction br depends on I_1, and I_1 depends on I_0. The helper thread in the figure consists of I_1 and br which is supposed to be a delinquent instruction.

When the instruction I_0 is fetched, the helper thread is triggered and the processor

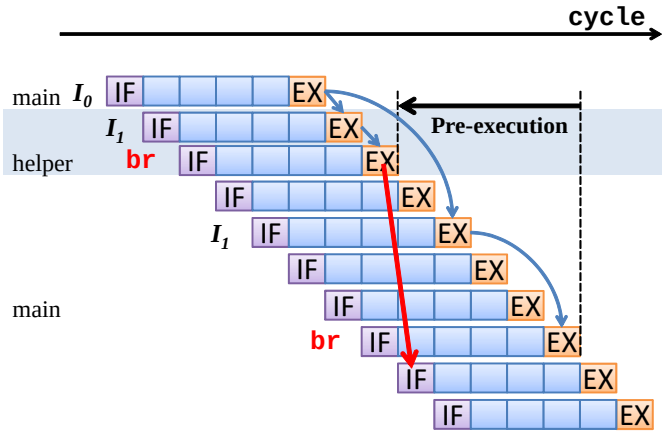

Figure 3.2: Pipeline of Helper Threading

starts to fetch it. After that, when br in the main thread is fetched, the next instruction to be fetched is decided by the execution result of br in the helper thread. Thus, when the branch prediction is not confident, Helper Threading can use the result from the helper thread and predict quite accurately which instruction to fetch. This example hides the latency of the branch instruction, but Helper Threading can also hide latencies of cache misses in the same way as prefetching.

3.2.1 Detail of Helper Threading

In order to execute a delinquent instruction in a helper thread earlier enough than the real delinquent instruction in the main thread, instructions of the helper thread must be increased to be triggered earlier. However, executing the helper thread consumes processor's resources, so the helper thread interrupts execution of the main thread if the number of instructions in the helper thread is large. Therefore, it is important to decide how to create and execute the helper thread.

When to Create a Helper Thread There are two ways to create a helper thread: a compiler creates statically with a profile, or a processor creates dynamically when executing the program. *Simultaneous Subordinate Microthreading (SSMT)* [1] and *Speculative Data-Driven Multithreading (DDMT)* [15] are examples of the former method. In these techniques, a compiler can perform complex optimization. However, this method cannot use dynamic information.

On the other hand, *Dynamic Speculative Precomputation* [2] creates a helper thread dynamically. This method can define the number of instructions of a helper thread by using dynamic information, but it cannot do complicated optimization.

How to Create a Helper Thread The techniques noted above implement the same method to create helper thread. When a delinquent instruction is identified by using profiles or miss events, find the instructions on which the delinquent instruction depends. Then, recursively find instructions which the instructions found previously depend on and finally create the helper thread. As we mentioned above, in this method, the key point is how many instructions to add to the helper thread.

Trigger to Start Helper Threading Among the techniques mentioned above, the static methods insert explicit thread starting instructions when compiling the program. In **figure 3.2**, the instruction I_0 corresponds to the thread starting instruction. On the other hand, the dynamic methods start a helper thread when the instruction in the main thread on which the first instruction in the helper thread depends is fetched. In this method, in figure 3.2, the source operand of I_1 depends on the destination operand of I_0.

3.2.2 Usage of the Result of a Helper Thread

The result of a helper thread is mainly used for branch prediction or prefetching. Additionally, in DDMT, the main thread does not execute instructions which a helper thread has executed, to reduce consumption of the resources of the processor [15] . However, this technique needs an extremely large table to gather execution results of the helper thread. Furthermore, the main thread still has to fetch all instructions.

3.2.3 Problem of Helper Threading

Helper Threading picks up a delinquent instruction and instructions it depends on and creates a helper thread. Then, when the trigger instruction is fetched, the helper thread starts to be fetched.

To get enough benefit from Helper Threading, a 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. This is because the helper thread is relatively large compared to the main thread. In addition, such a large extra thread consumes the resources of the processor, so the main thread is interrupted in execution. This problem is unavoidable as long as Helper Threading executes extra instructions and most delinquent instructions are executed in the small loops. Therefore, Helper Threading cannot achieve an effective improvement in performance.

Chapter 4

Overview of Switch-on-Future-Event Multithreading

We propose a technique of hiding latencies of delinquent instructions, called *Switch-on-Future-Event multithreading (SoF-MT)*. This technique regards each iteration of a loop as a thread and executes them simultaneously with an SMT processor. 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, SoF-MT switches the thread from one iteration to another, triggered by a fetch of a delinquent instruction. Therefore, SoF-MT can hide the latency without suffering from the problem.

4.1 Multithreading Model

To execute each iteration of a loop simultaneously with an SMT processor, we add some special instructions to the instruction set.

4.1.1 Beginning and End of Threads

Figure 4.1 shows how a loop is extracted to threads. SoF-MT starts simultaneous multithreading when a *pstart* instruction is executed. **Figure 4.2** shows the allocation of threads on the number of n logical processors in the SMT processor. In the figure, t_i shows the thread which corresponds to the i th iteration. At the beginning of the multithreading, 0 th to $i-1$ th threads are created on corresponding logical processors.

Each thread ends with a *pend* instruction. When a pend is committed, the thread right after the number of the logical processor will be created. In **figure 4.2**, for example, when the logical processor 0 has committed pend in the thread t_0, it creates a new thread of t_n on itself.

An instruction *pexit* shows the end of the loop. When a pexit is committed, the processor exits the loop and fetches subsequent instructions with single threading.

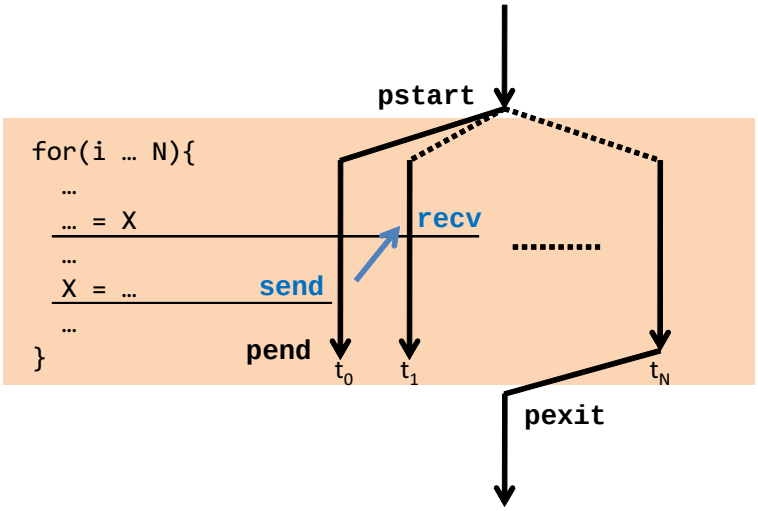

Figure 4.1: Multithreading Model of SoF-MT

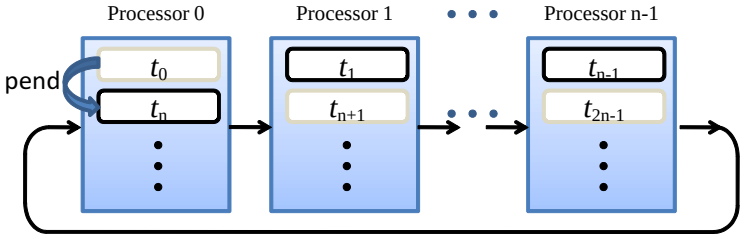

Figure 4.2: Allocating Threads to n Logical Processors

These additional instructions are inserted statically by the compiler.

4.1.2 Solution of Carry-forward Dependencies

Each logical processor is connected with a unidirectional ring and can communicate with a neighbor processor. As we show in figure 4.2, if there are any carry-forward dependencies of registers among iterations, the logical processor i sends data to the logical processor $(i+1)\%n$, ($\%$ indicates a modulus operator) to resolve these dependencies. We propose instructions *send* and *recv* to tell when a logical processor should send and receive data. These instructions are also inserted by the compiler.

If a register never changes its value in a loop, instructions send and recv that correspond to the register are not inserted. This considerably reduces the need to insert send and recv instructions. Furthermore, these registers can map to the same physical registers among threads, so the logical processors can share the resources of the physical registers.

4.2 How to Hide Latencies of Delinquent Instructions

In SoF-MT, when an instruction is fetched, the processor predicts whether it will cause any "misses" at the timing of execution. If it is determined that it will cause a miss, the processor switches the thread to fetch before the actual miss occurs.

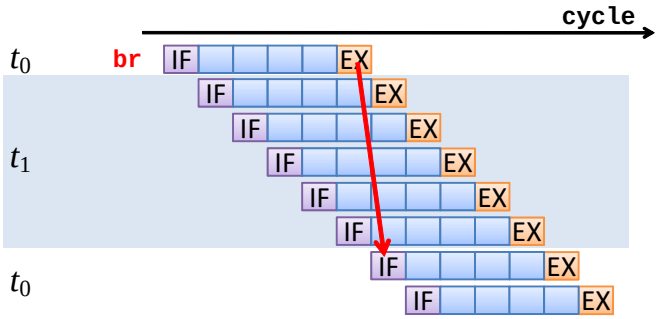

Figure 4.3: Pipeline of SoF-MT

For instance, **figure 4.3** shows hiding of the latency of a branch instruction. In this figure, the instructions of the pipeline chart where the background is not painted belong to thread t_0, while those where the background is painted belong to thread t_1. This figure shows a branch instruction br of t_0 which is predicted as a delinquent instruction. In this situation, in the next cycle the processor fetches t_1 and keeps executing the program. When the br has been executed, the processor switches back to the thread and fetches instructions of t_0. Thus, in SoF-MT, the processor hides the latency of br without using branch prediction. This is an example of a branch instruction, but SoF-MT can do the same thing for load instructions. When it fetches a load instruction that seems to miss the cache, it switches the thread. This prevents instructions after the load instruction from being stucked in the processor.

Chapter 5

Detail of Implementation

SoF-MT has to guarantee that the result of its execution is the same as that of sequential execution. Consequently, in general, instructions of the following iterations cannot be executed until all the precedent iterations have finished executing their instructions. In this situation, if instructions of the following threads are fetched, they get stuck in the instruction window and degrade performance of the processor. To make matters worse, the instruction window of the processor may be completely filled with these instructions and cause deadlock. To avoid this problem, we propose optimal control of thread switching and speculative retirement.

5.1 Control of Thread Switching

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

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

5.1.1 Learning and Detecting Delinquent Instructions

The processor predicts whether an instruction is a delinquent instruction by accessing tables shown in **figure 5.1**. Figure 5.1 (1) is for branch instructions, while (2) is for load instructions.

Prediction of Branch Prediction Miss First, we take a look at **figure 5.1** (1).

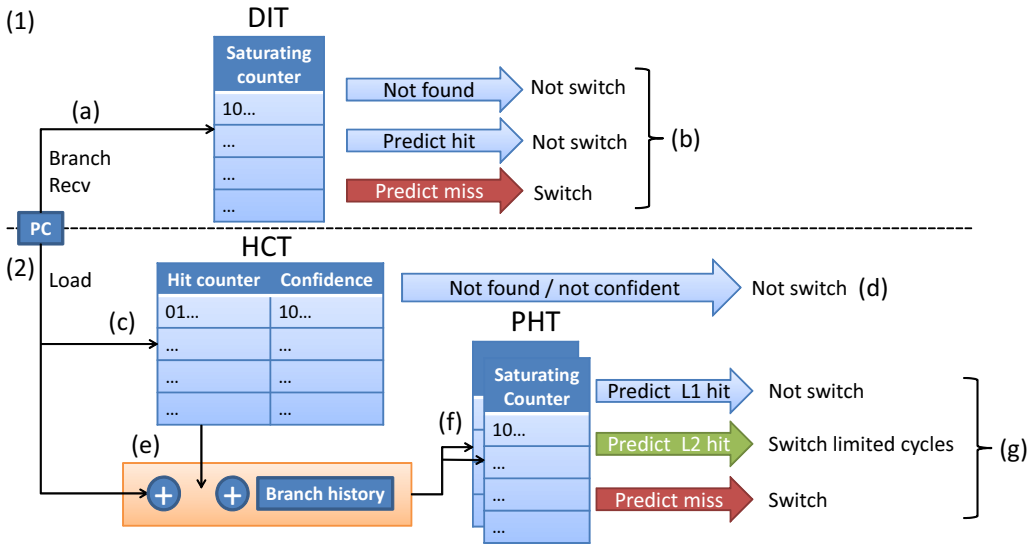

Figure 5.1: Prediction of Delinquent Instructions

To predict a branch prediction miss, we use saturation counters in the table called the *Delinquent Instruction Table (DIT)*. DIT is a set-associative tagged table indexed by the PC of a branch instruction (a).

Learning of delinquent instructions with the DIT is done as follows. When a branch instruction has committed, if the branch predictor has mispredicted, the counter in the DIT is incremented. If there is no corresponding entry, allocate one and clear to zero. If the branch prediction hits and there is a corresponding entry in the DIT, the counter is decremented. On predicting whether a branch is delinquent, when the processor has fetched the instruction, it reads the DIT and if the counter is above the threshold level, it detects the instruction as a delinquent instruction. If there is no entry, or the counter is below threshold level, the instruction is not determined as a delinquent instruction (b).

The point of this technique is that the DIT works well even if it is small. As we mentioned in Section 2, delinquent instructions are a small number of static instructions. Furthermore, most branch instructions do not miss the branch prediction and never consume entries of the DIT. Therefore, the DIT works effectively with a small number of entries.

Prediction of Periodic Cache Miss A predictor of cache hit/miss is implemented in some processor architectures. For example, the Alpha 21264 microprocessor architecture adopts a load hit/miss predictor in order to optimize issue timing of instructions that are dependent on a load instruction [11]. The predictor on Alpha

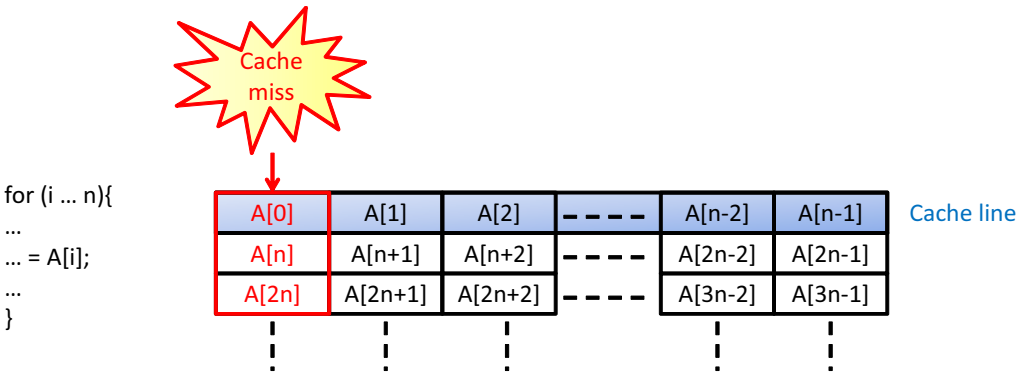

Figure 5.2: Periodic Cache Miss

21264 is the MSB of a 4-bit saturating counter which decrements by 2 when there is a load miss, otherwise it increments by 1 when there is a hit.

However, such a predictor sometimes does not work effectively because it cannot predict periodic cache misses such as array accesses using a loop variable. In **figure 5.2**, for instance, an array $A[i]$ is read in a *for* loop using a loop variable. The size of a cache line is the same as the size of n elements of the array. In this situation, obviously the load instruction that accesses the array misses the cache every n time, but the predictor cannot predict such cache misses. Consequently, we propose a new cache hit/miss prediction system for the SoF-MT shown in figure 5.1 (2).

There are two or more tables in the system.

Hit Count Table (HCT) is a set-associative tagged table indexed by the PC of a load instruction, where the entry stores the number of consecutive cache hits and the confidence of prediction (c). In the learning phase, the hit count is incremented when a load instruction hits the cache, otherwise it resets to zero when the load misses the cache. The confidence counter is incremented when the prediction hits, otherwise it is decremented. When the prediction is determined unreliable or the entry is not found, the thread will not be switched because a load instruction often hits the cache (d). In the same way as DIT, an entry of a static instruction is allocated to the HCT only when it misses the cache. Therefore, HCT can be small to get sufficient performance.

On the other hand, *Pattern History Table (PHT)* is a direct-map untagged table. It is indexed by a convoluted number of the PC, the global branch history and the cache hit count produced by the HCT (e), (f). This enables access of the same index when a load instruction hits the cache consecutively. We also convolute the branch history to deal with multiple loops. The entry of PHT is a saturation counter, which increments when a load instruction misses the cache, otherwise decrements

when it hits. The MSB of PHT is a prediction result whether a load will miss or not. The number of PHTs is the same as the number of levels of caches. PHTs are accessed simultaneously and referred or updated if corresponding caches miss. We implement timers to the number of the logical processors to count how many cycles they have to wait until the data reaches the L1 cache.

5.1.2 Detection of Deadlocks and Recovery

A deadlock occurs when recv and its dependent instructions fill the instruction window and the paired send cannot be put into the instruction window.

The usual case is that all the instructions in the instruction window are waiting to finish executing other instructions and are not selected. Therefore, we cannot say that none of the instructions are selected so the deadlock occurs. We call this usual case *Break of Selection*.

The difference between Break of Selection and deadlock is the number of *executing* instructions. The *executing* instructions stand for instructions that have been selected and issued, but not yet committed. In case of Break of Selection, all the instructions in the instruction window wait for *executing* instructions, while in case of a deadlock, they wait for a send instruction which is not in the instruction window.

We focus on this difference to detect a deadlock. If there is no *executing* instruction, the instruction window is full, and no instruction is selected, the processor detects a deadlock. When it detects the deadlock, all the threads apart from the most preceding thread are flushed.

5.2 Speculative Retirement

To guarantee execution order naively, the threads, other than the most preceding thread, cannot retire instructions. However, this leads not only to degradation of performance but also to the deadlock in the worst case, as we noticed above.

Therefore, in SoF-MT, the threads other than the most preceding thread perform *Speculative Retirement*, which is to retire instructions from the processor as if they had nothing to do with other threads. As instructions of a speculative thread retire, we have to keep consistency of memory access order between threads. From now, we call the most preceding thread a *non-speculative thread*, while other threads are *speculative threads*. Furthermore, we call memory access order violation between threads as a *conflict* and a reversion of the execution result of the thread as an *abort*.

5.2.1 Keeping Consistency of Memory Access Order

To guarantee the result of an execution, if a memory instruction causes a conflict, the processor should abort the thread. We detect this conflict and abort with a system similar to *Transactional Memory* [6]. There are two differences between Transactional Memory and SoF-MT. One is that the threads of SoF-MT have priority, and the other is that although the original Transactional Memory manages

memory accesses by a *cache line* unit, we manage memory accesses by a *byte* unit.

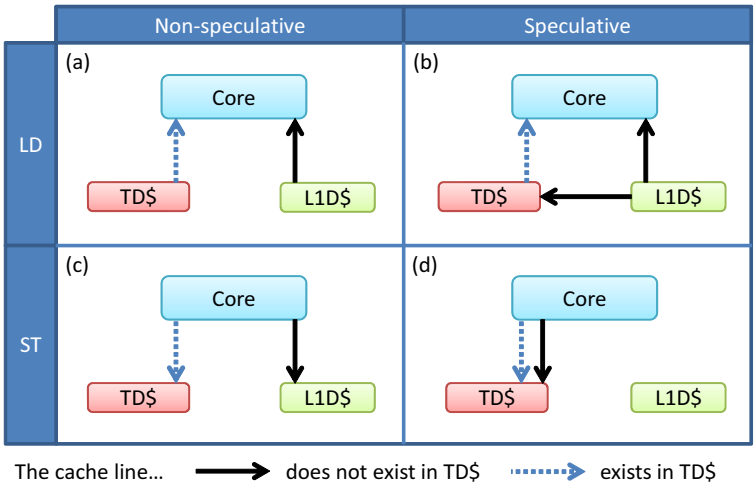

Figure 5.3: Behavior of Transactional Data Cache

As we mentioned in figure 5.2, the threads of SoF-MT often access a cache line sequentially. So, if SoF-MT manages memory accesses by a cache line unit, false sharing occurs frequently. However, if SoF-MT manages memory accesses by a byte unit, the number of additional bits gets increased compared with the data. This considerably increases the size of the cache and results in performance degradation. To avoid this, we propose a much smaller cache to store speculative data and detect conflict, called the *Transactional Data Cache*.

Transactional Data Cache is a cache which the processor can access in parallel with the L1 cache and can access by the same latency of the L1 cache, but is much smaller than the L1 cache. The difference between Transactional Data Cache and the L1 cache is that Transactional Data Cache has an invalid bit, a dirty bit, *Speculative Read bits (SR bits)* and *Speculative Write bits (SW bits)* in each byte. The number of SR bits and SW bits are the same as the number of logical processors. These bits indicate that a thread has read/written data from/to that area, respectively. Each logical processor checks these bits when they access a cache line and if any bit is set, the processor detects conflict and aborts the following threads. There are two exceptions to this rule. First, when a thread accesses Transactional Data Cache to load the data and if there are only SR bits set, an abort does not occur because there is no inconsistency. Second, the SR bit and the SW bit of the non-speculative thread are always unset because a memory access of the thread is always non-speculative.

Figure 5.3 shows how a thread accesses the Transactional Data Cache and the L1 cache. First, we take a look at memory reads. If a thread is the non-speculative

thread, it reads the data from the Transactional Data Cache if there is valid data. If not, it reads the data from the L1 cache. On the other hand, if the accessing thread is a speculative thread and the data is not in Transactional Data Cache, the Transactional Data Cache refills the cache line and sets corresponding bits. Next, we take a look at memory writes. If a thread is the non-speculative thread, it writes the data to the Transactional Data Cache if there is valid data. If not, it writes the data to the L1 cache. On the other hand, if the accessing thread is a speculative thread, it always writes the data to Transactional Data Cache. Remember that if a cache line with any bit is set is evicted from the Transactional Data Cache, it is considered as a conflict and some speculative threads are aborted to keep consistency.

In this technique, if there is no data in the Transactional Data Cache, the non-speculative thread does not access to the Transactional Data Cache. This considerably reduces cache line eviction and aborts. SoF-MT basically executes non-speculative thread, so the Transactional Data Cache gets smaller to achieve performance.

To avoid those violations occurring frequently, the processor adds the instruction just before the instruction which causes violation to DIT. The WDF of the thread that stops with this delinquent instruction is not unset until a pend instruction of the preceding thread has retired.

5.2.2 Value Prediction of recv Instructions and Delinquent recv Instructions

Although most of the time an iteration variable changes monotonically as an iteration goes on, send and recv are inserted for the variable because it is not loop invariant. A recv and instructions which depends on it cannot be executed until the corresponding send is committed, which results in performance degradation.

Therefore, we propose *Value Prediction for recv Instructions* using a stride predictor [17, 9]. The stride predictor adds the last value and the difference between the last value and the value before the last to predict the next value. Additionally, it does not predict the next value when the stride changes its value to keep confidence. Thus, it is effective to predict a variable that changes its value monotonically. This prevents recv from retirement and as a result subsequent instructions cannot retire.

Verification of Prediction Results Using Registers The original Stride Value Predictor verifies its prediction when an instruction is executed. However, in value prediction of recv instructions, the instructions should be retired from the processor to give up the entry of the instruction window. Therefore, in our scheme, the processor retires recv speculatively and keeps the predicted value in the register. Verification of the prediction is executed when the corresponding send is committed. If the processor finds that the prediction is incorrect, abort recv and the subsequent instructions and following threads.

Delinquent recv Instructions The predictor cannot predict values of recv instructions that do not change monotonically. Such recv instructions keep following instructions to the instruction window like delinquent load instructions. We regard these instructions as delinquent and manage them with DIT (figure 5.1 (1)).

Chapter 6

Evaluation

6.1 Evaluation Environment

We evaluated the technique using the following models on a cycle-accurate processor simulator developed in our laboratory. Parameters for the evaluation are shown in **table 6.1**. As the number of threads in the SMT processor increases, we have to increase the number of entries of the register files, too. **Table 6.2** shows the number of entries in the register files that corresponds to the number of threads.

We modified the compiler gcc-4.3.3 to insert additional instructions as mentioned in 4.1. This compiler can select whether to insert those instructions or not, and we compiled binaries for the baseline model and the helper threading model without them. We used the compile option "-O3" to compile benchmarks. **Table 6.3**

Table 6.1: Simulation Configurations

Name	Parameter	Name	Parameter
ISA	Alpha 21164A	miss penalty	10 cycle
fetch width	4 inst.	BTB	2K entry, 4-way
execution unit	int : 2, fp : 2, mem : 2.	L1C	32KB, 4-way, 64B/line, 3 cycle
instruction window	int : 32, fp : 16, mem : 16	L2C	4MB, 8-way, 64B/line, 15 cycle
branch prediction	8KB g-share	main memory	200 cycle

Table 6.2: Number of Entries of Register Files Corresponds to the Ways of SMT

Ways of SMT	Register Files
Single Thread (Baseline)	int: 128, fp: 128
2 Threads	int: 128, fp: 128
4 Threads	int: 256, fp: 256
8 Threads	int: 512, fp: 512
16 Threads	int: 1024, fp: 1024

Table 6.3: Evaluated Benchmarks

Benchmark Sets	Applications
SPECCPU 2006[19]	perlbench, mcf, milc, gobmk, hmmer, lbm
MediaBench[8]	adpcm_dec, adpcm_enc
EEMBC[3]	dither

shows the benchmarks we used for evaluation. We selected the benchmarks for which delinquent instructions are obvious and inserted the instructions to the corresponding loops using dedicated pragma for SoF-MT. Although, we used all 29 benchmarks of SPECCPU 2006 when deciding parameters of Helper Threading in Section 6.2.1 because we do not have to choose loops and insert the instructions for SoF-MT.

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

6.1.1 Evaluation Models

We evaluated the following models:

Baseline The baseline model with single thread execution.

Helper A model of Helper Threading. A helper thread is created dynamically in this model. **Table 6.4** shows additional hardware for the Helper Threading we implemented. We used DIT to find both delinquent branches and loads because there are no descriptions of how to find them in the articles on Helper Threading. We implemented a Retired Instruction Buffer which keeps retired instructions to find the instructions a delinquent instruction depends on.

SoF A model of SoF-MT. **Table 6.5** shows additional hardware for SoF-MT. Each entry of the DIT increments by n when the branch prediction misses, while it decre-

Table 6.4: Parameters of Helper Threading

Name	Parameter
DIT	1KB, 4-way, 3 bits/count
Retired Insn. Buffer	1K entry

Table 6.5: Parameters of SoF-MT

Name	Parameter
DIT	1KB, 4-way, 3 bits/count
HCT	160B, 2-way 8 bits/count 2 bits/conf
PHT	256B, 2 bits/count
Transactional Cache	16-way, 64B/line, 3 cycle 2KB, 4KB, 8KB, 16KB, 32KB
Stride Value Predictor	32 entries, CAM
Additional insn. window	send/recv:32

ments by 1 when it hits. The number n is determined in Section 6.2.2. The instruction is regarded as a delinquent instruction when the entry of the instruction is not 0. On the other hand, each entry of the PHT increments by 1 when a load instruction misses the cache, otherwise it decrements by 1 when it hits the cache.

6.2 Results

6.2.1 Configuration of Helper Threading

As we mentioned in Section 3.2, it is important how to create and execute the helper thread. So, first, we evaluate to determine parameters of Helper Threading. We focus on 3 parameters as follows and evaluate to achieve the best performance with Helper Threading.

Maximum instructions in a helper thread First, we explore the number of instructions in a helper thread at a maximum. Although we can execute a delinquent instruction in the helper thread earlier when this value is large, the helper thread consumes more resources of the processor and may cause performance degradation.

Figure 6.1 shows the IPCs relative to the baseline. The graph changes the maximum number of instructions in a helper thread while the increment number of DIT is fixed at 3 and the number of threads is 4. The processor considerably improves its performance in *462.libquantum* when the maximum number of instructions in helper threads is not less than 32 instructions. On the other hand, some applications such as *434.zeusmp* and *dither* significantly degrade their performance when the maximum number of instructions in a helper thread increases.

Increment number of DIT Second, we evaluate how many times we increment the entry of DIT when a branch miss or a cache miss occurs. To predict more delinquent instructions, this value should be larger. However, false predictions also get larger.

Figure 6.2 shows IPCs relative to the baseline with 1 to 6 increment numbers for DIT. The maximum number of instructions in helper threads is fixed at 16 and the number of threads is 4. The processor improves its performance in some applications such as *433.milc* and *450.soplex* by increasing the number how many times the entry of DIT increments. However, it do not change or degrade its performance in other benchmarks when increasing the value.

Number of threads Finally, we determine how many threads the processor can execute simultaneously. If the distance between delinquent instructions is short, the processor has to allocate threads for each delinquent instruction and execute them simultaneously. However, an increase in helper threads may results in performance degradation.

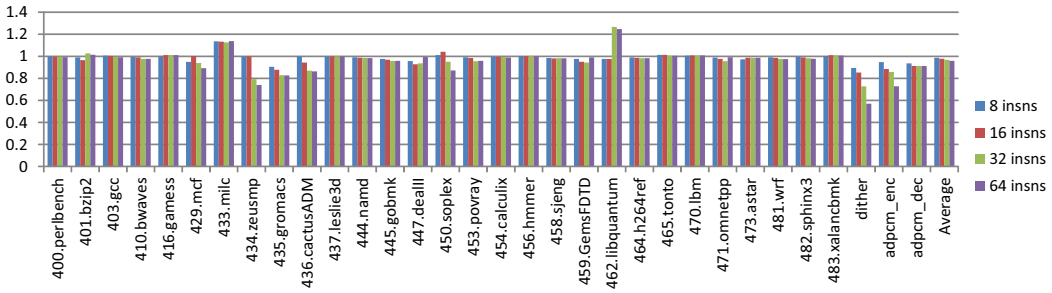

Figure 6.1: Relative IPC of Helper Threading when changing the maximum number of helper threads

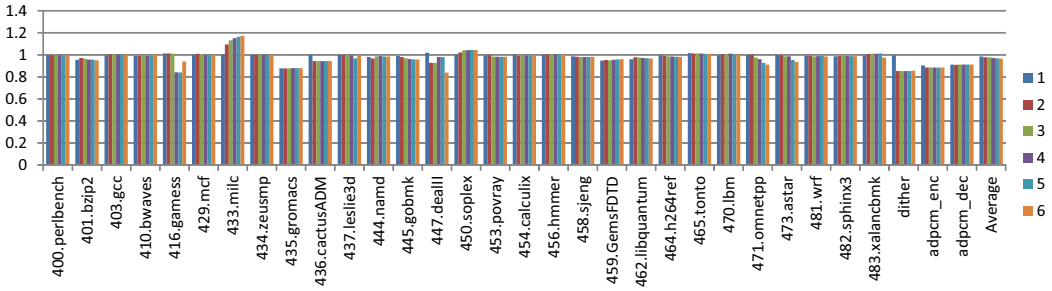

Figure 6.2: Relative IPC of Helper Threading when changing the increment number of DIT

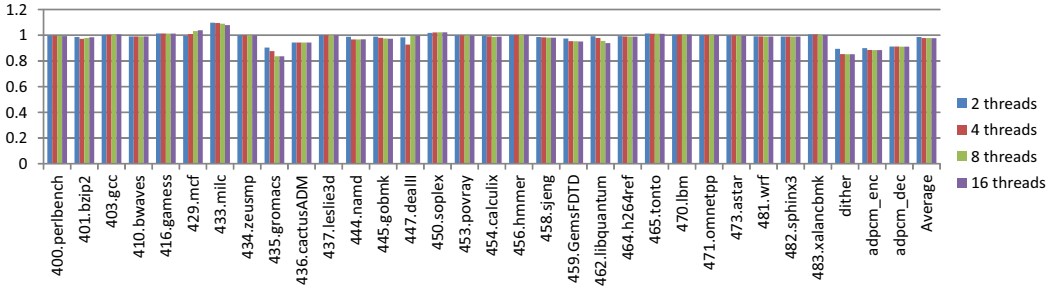

Figure 6.3: Relative IPC of Helper Threading when changing the number of threads

Figure 6.3 shows IPCs relative to the Baseline. The graph changes the number of threads in the processor while maximum number of instructions in a helper thread is 16 and the increment number of DIT is 3. The processor slightly improves its performance in *429.mcf* by increasing threads, while it degrade its performance in *435.gromacs* and *dither* .

To summarize the evaluation results, it is found that there are trade-offs to all three parameters and cannot decide which is the best value. Hereafter, we fix the maximum number of instructions in a helper thread at 16, the increment number of DIT at 3 and the maximum number of threads at 4.

6.2.2 Configuration of SoF-MT

Similar to Helper Threading, SoF-MT has some parameters to determine. In this section we fix the increment number of DIT and the number of threads for SoF-MT. Note that send and recv instructions are **NOT** included in numerators of IPCs to keep fairness.

Increment number of DIT **Figure 6.4** shows IPCs relative to the baseline with 1 to 6 increment numbers for DIT. The results show that the increment number other than 1 does not have an effect on performance improvement. The processor achieves the best performance improvement when the increment number of DIT is 1. Therefore, hereafter, we fix the increment number of DIT at 1.

Number of threads **Figure 6.5** shows IPCs relative to the baseline when changing the number of threads in the processor. Unlike Helper Threading, when the number of threads increases, the performance tends to improve. The reason for this is that the processor can hide latencies of delinquent instructions more efficiently when the number of thread increases. However, on *470.lbm*, the processor degrades its performance significantly when the number of thread is 16. We will discuss the reason for this in Section 6.2.4. Hereafter, we fix the number of threads at 16.

6.2.3 Conclusive Performance Improvement

Figure 6.6 shows the relative IPCs of Helper Threading and SoF-MT to the baseline. Each legend "SoF-nKB" shows the SoF-MT model with the size of Transactional Data Cache is fixed to nKB. The Helper Threading model improves its performance in some benchmarks and at a maximum of 13.2% in *433.milc*.

On the other hand, when increasing the size of Transactional Data Cache in SoF-MT, *429.mcf* and *433.milc* improve their performance. We can say that they have many memory access instructions in the target loop and consume Transactional Data Cache. Even so, they achieve sufficient performance improvement when the size of Transactional Data Cache is 4KB. The results show that SoF-MT achieves performance improvement by an average of 10.1% and a maximum of 38.7% in *adpcm_dec*.

Figure 6.7 shows the proportion of the flushed instructions to the fetched instructions. The results show that although Helper Threading prevents the processor from fetching instructions to be flushed, SoF-MT achieves much better effects not to fetch such instructions. Especially, in *adpcm_dec*, Helper Threading reduced the flushed instructions from 45.2% to 41.0%, while SoF-MT reduced to 5.4%.

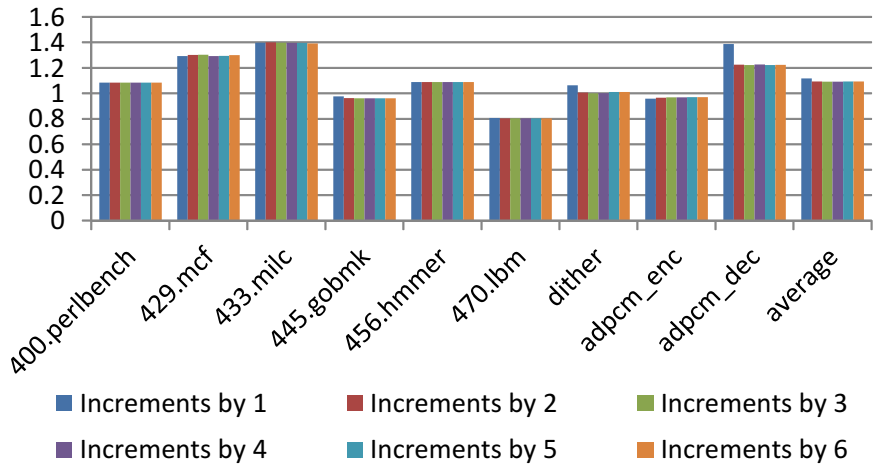

Figure 6.4: Relative IPC of SoF-MT when changing the increment number of DIT

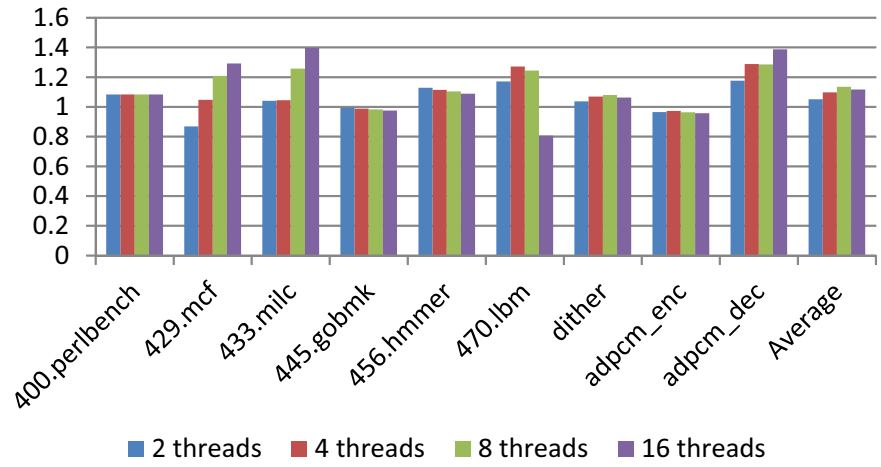

Figure 6.5: Relative IPC of SoF-MT when changing the number of threads

Figure 6.8 shows the proportion of the executed instructions in helper threads and additional instructions of SoF-MT to all executed instructions. In most cases SoF-MT executes fewer extra instructions and provides better performance improvement than Helper Threading. Moreover, even in *456.hmmer* and *adpcm_dec* in which SoF-MT executes more extra instructions than Helper Threading, the processor achieves better performance improvement from SoF-MT. This result proves that SoF-MT hides latencies of delinquent instructions more effectively than Helper Threading.

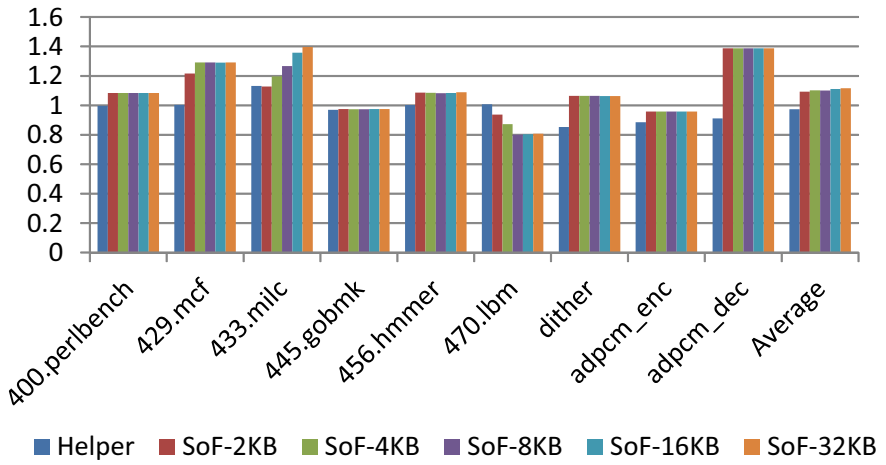

Figure 6.6: IPC Relative to the Baseline

6.2.4 The Number of Threads and Instruction Window

In **figure 6.6**, the performance degradation occurs on *470.lbm* when the size of Transactional Data Cache increases. However, the number of flushed instruction is decreasing when the size of Transactional Data Cache increases as you can see in **figure 6.7**.

The reason for this is as follows. When the Transactional Data Cache is small, aborts occur frequently because of cache line evictions. On the other hand, when the Transactional Data Cache is large, such evictions do not occur so frequently. However, if the number of entries in an instruction window for memory access instructions is close to or smaller than the number of threads, delinquent loads fill the instruction window and cause pipeline stalls of the frontend. In the model of smaller Transactional Data Cache, the processor incidentally avoids these stalls by aborting threads and results in preventing performance degradation. Figure 6.5 proves this logic that the processor degrades its performance only when the number of threads is 16, which is the same number of the entries in the instruction window.

6.3 Evaluation on Wider Memory Instruction Window

We evaluate with the model that the instruction window of memory instructions is larger than the previous model. The processor can store 32 instructions in the instruction window in this model.

Figure 6.9 shows the relative IPCs of Helper Threading and SoF-MT to the single thread execution. The results show that the processor achieves better performance than the previous model on *470.lbm*. The other benchmarks show almost the same

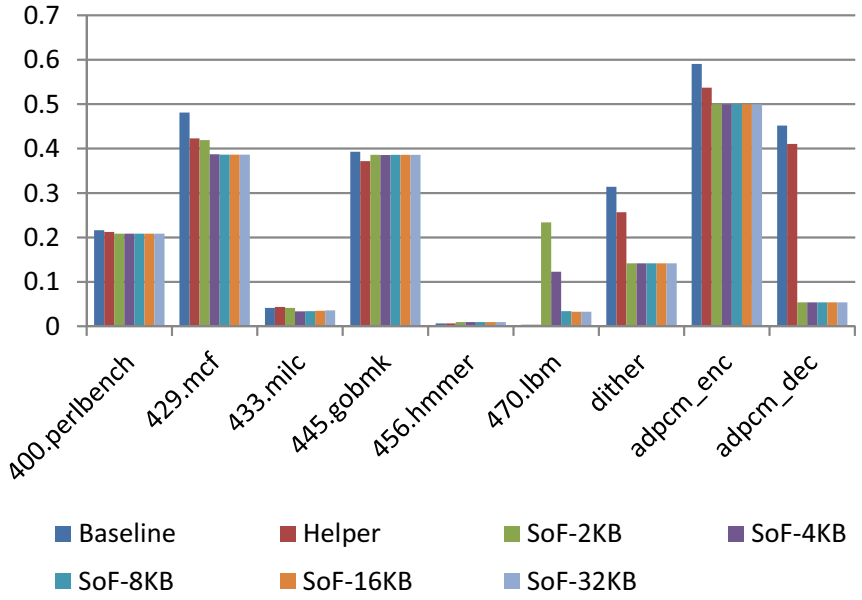

Figure 6.7: Proportion of Flushed Instructions to Fetched Instructions

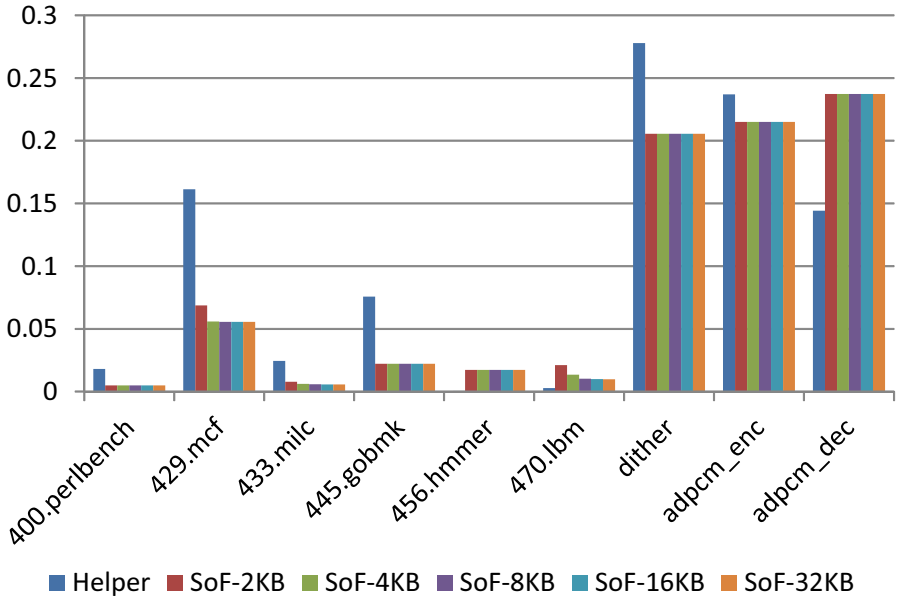

Figure 6.8: Proportion of Executed Extra Instructions to All Executed Instructions

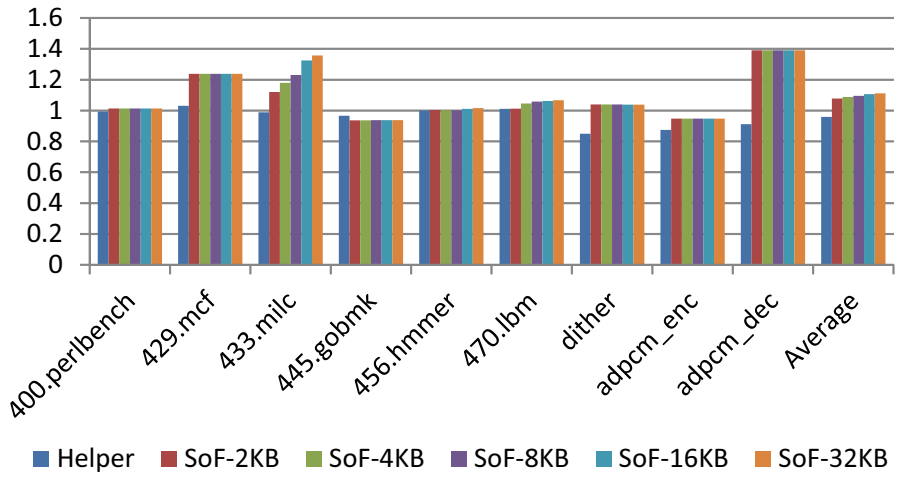

Figure 6.9: IPC Relative to the Baseline with Wider Instruction Window for Memory Instructions

results as the previous model and SoF-MT achieves performance improvement by an average of 10.9% and a maximum of 39.0% in *adpcm_dec* when the size of Transactional Data Cache is 4KB.

Chapter 7

Related Work: Throughput-Oriented Multithreading

A Throughput-Oriented Multithreading processor basically executes independent programs simultaneously to maximize total throughput of the processor. A typical research of Throughput-Oriented Multithreading for delinquent instructions is *Switch-on-Event Multithreading* [5, 12]. This technique switches a thread to another when a load instruction misses the cache in order to hide the latency of memory access. Besides, there are some methods for SMTs to speed up throughput of the processor. For example, Eyerman et al. proposed a fetch policy to find a long-latency load and switch the thread so as not to fetch instructions which eventually stall [4]. Luo et al. proposed a fetch policy not to fetch a thread which has many unresolved branches that are predicted with low confidence. This enables to avoid fetching instructions that have high possibility to be flushed [10].

As we mentioned before, Throughput-Oriented Multithreading basically executes independent programs simultaneously. Therefore, they improve total execution throughput of the processor by hiding the latency of delinquent instructions, but they cannot speed up each single program.

Chapter 8

Conclusion

In this paper, we proposed SoF-MT which hides the latency of delinquent instructions. SoF-MT regards each iteration of a loop as a thread and executes them simultaneously. The processor switches a thread when it fetches a delinquent instruction to hide the latency of a miss that the instruction will cause in the future. This technique works well because delinquent instructions are in small loops and the processor can create a sufficient number of threads to switch. Our evaluation shows that SoF-MT achieves performance improvement by an average of 10.1% and a maximum of 38.7% from the single thread execution, while the conventional Helper Threading provides only 13.2% speedup at a maximum.

We have evaluated only 9 programs for SoF-MT because of the restriction of the compiler. We will explore the scheme inserting the instructions for SoF-MT automatically. We also expect to design a real processor which SoF-MT is implemented and evaluate on it.

Bibliography

[1] R. Chappell, J. Stark, S. R. S. Kim, and Y. Patt. Simultaneous subordinate microthreading (ssmt). In *ISCA*, pages 186–195, May 1999.

[2] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *ISCA*, pages 14–25, 2001.

[3] The Embedded Microprocessor Benchmark Consortium. *http://www.eembc.org/*.

[4] S. Eyerman and L. Eeckhout. A memory-level parallelism aware fetch policy for smt processors. In *HPCA*, pages 240–249, Feb. 2007.

[5] M. Farrens and A. Pleszkun. Strategies for achieving improved processor throughput. In *ISCA*, pages 362–369, 1991.

[6] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, May 1993.

[7] D. Joseph and D. Grunwald. Prefetching using markov predictors. *Computers, IEEE Transactions on*, 48(2):121–133, Feb 1999.

[8] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *MICRO*, pages 330–335, Dec 1997.

[9] M. Lipasti and J. Shen. Exceeding the dataflow limit via value prediction. In *MICRO*, pages 226–237, Dec 1996.

[10] K. Luo, M. Franklin, S. Mukherjee, and A. Sezne. Boosting smt performance by speculation control. In *International Parallel and Distributed Processing Symposium*, page 9, Apr 2001.

[11] E. Mclellan and D. Webb. The alpha 21264 microprocessor architecture. In *ICCD*, page 90, October 1998.

[12] C. McNairy and R. Bhatia. Montecito: a dual-core, dual-thread itanium processor. *Micro, IEEE*, 25(2):10–20, March-April 2005.

[13] T. Ohsawa, M. Takagi, S. Kawahara, and S. Matsushita. Pinot: Speculative multi-threading processor architecture exploiting parallelism over a wide range of granularities. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, Washington, DC, USA, 2005. IEEE Computer Society.

[14] S. Palacharla and R. Kessler. Evaluating stream buffers as a secondary cache replacement. In *ISCA*, pages 24–33, Apr 1994.

[15] A. Roth and G. Sohi. Speculative data-driven multithreading. In *HPCA*, pages 37–48, 2001.

[16] K. RYOTARO, O. YUKIHIRO, I. MITSUAKI, A. HIDEKI, and S. TOSHIO. A multiprocessor architecutre sky that exploits thread-level parallelism in non-numerical applications (special issue on multimedia network system). *Transactions of Information Processing Society of Japan*, 42(2):349–366, 2001-02-15.

[17] Y. Sazeides and J. Smith. The predictability of data values. In *MICRO*, pages 248–258, Dec 1997.

[18] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *ISCA*, pages 414–425, Jun 1995.

[19] The Standard Performance Evaluation Corporation. *SPEC CPU2006 suite http://www.spec.org/cpu2006/*.

[20] T. SUNAO, K. MASAKI, M. MASATO, I. AKIHISA, K. AKIHIKO, and N. NAOKI. On-chip control parallel multi-processor: Muscat(special issue on parallel processing). *Transactions of Information Processing Society of Japan*, 39(6):1622–1631, 1998-06-15.

Publications

International Conferences

1. (pending) Naruki Kurata, Ryota Shioya, Masahiro Goshima and Shuichi Sakai: Switch-on-Future-Event Multithreading, The 39th International Symposium on Computer Architecture (ISCA), (2012)

Domestic Symposiums(with peer review)

1. (pending) Mitsuo Date, Naruki Kurata, Ryota Shioya, Masahiro Goshima and Shuichi Sakai: Processor Architecture that Minimizes Register Renaming and Dispatch Network, Symp. on Advanced Computing Systems & Infrastructures (SACSIS), (2012) (in Japanese).

2. (pending) Satoshi Arima, Naruki Kurata, Ryota Shioya, Masahiro Goshima and Shuichi Sakai: Timing-Fault-Tolerant Out-of-Order Processor, Symp. on Advanced Computing Systems & Infrastructures (SACSIS), (2012) (in Japanese).

3. (pending) Shuji Yoshida, Souichirou Hirohata, Naruki Kurata, Masahiro Goshima and Shuichi Sakai: A Clocking Scheme Enabling Dynamic Time Borrowing Processor, Symp. on Advanced Computing Systems & Infrastructures (SACSIS), (2012) (in Japanese).

4. Naruki Kurata, Ryota Shioya, Masahiro Goshima, and Shuichi Sakai: Improvement and Evaluation of Switch-on-Future-Event Multithreading, Symp. on Advanced Computing Systems & Infrastructures (SACSIS), Vol. 2011, pp. 82—91 (2011). (in Japanese).

5. Ryota Shioya, Naruki Kurata, Jun Nakashima, Masahiro Goshima, and Shuichi Sakai: Switch-on-Future-Event Multithreading, Symp. on Advanced Computing Systems & Infrastructures (SACSIS), pp. 157—165 (2010). (in Japanese).

Oral Presentations

1. Mitsuo Date, Naruki Kurata, Ryota Shioya, Masahiro Goshima, and Shuichi Sakai: Trace Cache Architecture which Eliminates Register Renaming and Dispatch Network, IPSJ SIG Technical Report 2011—ARC—196, (2011). (in Japanese).

2. Shuji Yoshida, Satoshi Arima, Naruki Kurata, Ryota Shioya, Masahiro Goshima, and Shuichi Sakai: Preliminary Experiment of A Clocking Scheme Enabling Dynamic Time Borrowing, IEICE Technical Reports CPSY2011—11, pp. 13—18 (2011). (in Japanese).

3. Mitsuo Date, Naruki Kurata, Yuji Ito, Ryota Shioya, Masahiro Goshima and Shuichi Sakai: Dispatched Image Cache, The 73rd National Convention of IPSJ, pp. 1-91—1-92 (2011). (in Japanese).

4. Naruki Kurata, Ryota Shioya, Jun Nakashima, Masahiro Goshima and Shuichi Sakai: An Improvement of Switch-on-Future-Event Multithreading, IPSJ SIG Technical Report 2010—ARC—190, No. 27 (2010). (in Japanese).

5. Naruki Kurata, Ryota Shioya, Masahiro Goshima and Shuichi Sakai: An Improvement of Branch Pre-Decision Focusing on Loops, The 72th National Convention of IPSJ, pp. 1-213—1-214 (2010). (in Japanese).

6. Naruki Kurata and Kazuhide Higuchi: Parallelization of O(ND) Algorithm by CUDA, Symp. on Advanced Computing Systems & Infrastructures (SACSIS), pp. 112-113 (2009) (poster). (in Japanese).

Awards

1. A multi-threaded processor targeted for loops, 72nd National Convention of IPSJ, Certificate of Excellent Undergraduate Thesis (2010).

2. Naruki Kurata and Kazuhide Higuchi: Won third prize in the category of the prescribed problem, GPU Challenge 2009 in conjunction with Symp. on Advanced Computing Systems & Infrastructures (SACSIS) (2009)

Acknowledgement

First of all, I would like to express my gratitude to my advisor, Professor Shuichi Sakai, for his supervision, advice, and encouragement through the three years I have spent in the laboratory. He provided me wonderful environment and opportunity to conduct my own research.

I truly acknowledge Associate Professor Masahiro Goshima for his invaluable comments for my research. I learned many things from his method of thinking about the research and other various things.

I am indepted to Ryota Shioya for his devoted support. He gave me many helpful suggestions and comments. My research could not have been done without his immeasurable help.

I would like to thank to the secretaries, Ms.Harumi Yagihara, who has helped me a lot on my application and to Ms.Tomoyo Ise and Ms.Tamaki Hasebe for their contribution to the lab.

Many thanks to the rest of all laboratory members for providing invaluable discussions and fun environment to learn. They help me in both public and private.

I also express my thanks to all my family and friends in the university of Tokyo and elsewhere for their support.