以投機狀況為基礎的同步多線程架構前端策略

Front-End Policy based on Speculation Condition for Simultaneous Multithreading Architecture

陳沛源
大同大學資訊工程所
g9306015@ms2.ttu.edu.tw

謝忠健
大同大學資訊工程所
shieh@ttu.edu.tw

摘要

對現代能夠進行多重發送的超純量處理器而言，指令提取單元是讓高效力的執行單元能夠保持全速運作的關鍵之一。用以評價指令提取單元的數據不只有發送指令的速率還有投機執行的準度。也就是說一個好的指令提取單元要能在合理的時脈時間內，從正確的執行路徑上擷取大量的指令。在同步多線程架構的處理器上狀況會有些許的不同，因為在處理器中同時有多個活動中的程序。若是能得知每個線程未來的投機執行狀況，前端的指令提取單元可以偏好具有高度可預測性執行路徑的執行緒，以避免誤入錯誤執行路徑時額外產生的資源及電力浪費。

在本論文中，我們將焦點擺在改善同步多線程處理器的前端執行單元。我們提出了一個輔助性的結構稱作 Sequential Trace Table (STT)，來提供對各個執行緒投機執行狀況的預先觀察。並利用這些投機執行狀況的資訊以輔助排定提取優先權的策略。

關鍵詞：同步多線程架構、提取策略、投機執行。

Abstract

For modern wide-issue superscalar processors, high performance instruction fetch unit is the key component to keep the powerful execution engine operating in full speed. The performance measurement to evaluate a front-end mechanism includes both the instruction delivery rate and speculation accuracy. That means a good front-end engine should be able to fetch and dispatch massive instructions on the right execution path, in a reasonable clock cycle time. Things may be a little different in Simultaneous Multithreading (SMT) architecture because there are multiple active contexts inside the CPU. If we can extract some information about future speculation conditions of each thread, the front-end fetch engine can then prefer threads with highly predictable execution path to avoid resource or energy waste on mis-speculative routes.

In this paper, we focus on improving the front-end engine of SMT processor. We present a supplementary structure called Sequential Trace Table (STT) to provide a look-ahead into the future speculating conditions of each thread, and use the information to help improving fetch prioritizing policies.

Key words: Simultaneous Multithreading Architecture, Fetch policy, Speculative Execution.

1. INTRODUCTION

Benefiting from the improving of semiconductor process, modern processor has much design space for advanced microarchitectures. At a high level view point, a modern high-performance processor is composed of two processing engines: the front-end processor and the execution core. The front-end processor is responsible for fetching and preparing (e.g., decoding, renaming, etc.) instructions for execution. The execution core orchestrates the execution of instructions and the retirement of their register and memory results to non-speculative storage. These two processing engines are decoupled with internal instruction storages and/or reorder structures, such as instruction queue, reorder buffer, and reservation stations.

For the execution part, the focus is on how to use the processor's resources effectively, by exploiting parallelism in the instruction stream provided by the front-end engine. Such efforts include out-of-order issue, speculative issue, value speculation, and Simultaneous Multithreading (SMT) execution engine. For the front-end part, the more and more powerful execution engine places further demands on improving both speed of instruction stream delivering and accuracy of execution path selection. This kind of researches result in sophisticated branch prediction schemas and fetch policies.

Unfortunately, fetch units implemented in most of the modern commercial microprocessors have rigid restriction on the range of instructions they can access in a single machine cycle:

1. The fetch unit can't fetch across a taken branch. That means the instructions fetched in one cycle often belong to the same basic block.
2. The fetch unit can't fetch across the boundary of a cache line. A multi-port instruction cache may help to mitigate this kind of limitation.

In this paper, we focus on improving the front-end engine of SMT processor. We present a supplementary structure called Sequential Trace Table (STT) to provide a look-ahead into the future speculating conditions of each thread, and use the information to help branch prediction and fetch prioritizing.

1.1 Simultaneous Multithreading

Multithreading in processor architectures is similar in concept to Preemptive Multitasking in operating systems but is implemented in a more fine-grained level based on modern out-of-order superscalar processors.

Simultaneous multithreading (SMT) is one of the two main implementations of multithreading, the other form being temporal multithreading. Figure 1.1 illustrates the characters of different kinds of multithreading. In Temporal Multithreading, only one thread of instructions can execute in any given pipeline stage at a time. In Simultaneous Multithreading, instructions from more than one thread can be executing in any given pipeline stage at a time. This is done without great changes to the hardware structure of modern wide-issue superscalar processors. The main additions needed are the ability to fetch instructions from multiple threads in a cycle, the ability to distinguish instructions from different threads, and a larger register file to hold the machine states of multiple threads. The number of concurrent threads can be decided by the chip designers, but practical restrictions on chip complexity have limited the number to two for most commercial SMT implementations.

In processor design, there are two ways to increase on-chip parallelism with less resource requirement: one is superscalar technique which tries to increase Instruction Level Parallelism (ILP), the other is multithreading approach exploiting Thread Level Parallelism (TLP). The SMT inherits all characteristics of superscalar architecture, so it is able to exploit ILP as well as TLP.

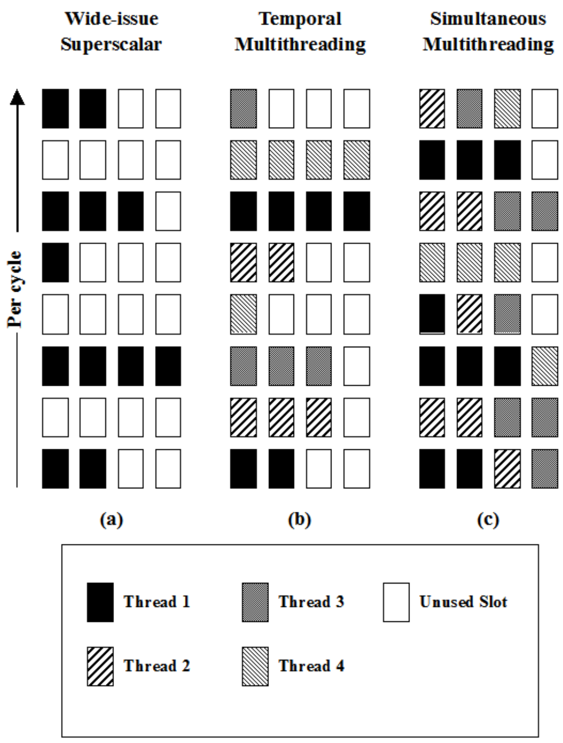

Figure 1.1: Comparison of different kinds of Multithreading.

1.2 Biased Branch

Some branch instructions show significant tendency to be mostly taken (branch out) or not-taken (fall through) in the instruction flow. We call this kind of branches "strongly biased branch". Most of the time, a branch categorized as Not-taken biased will not change the direction of execution route, so the instruction fetch unit can continue without being stopped. In contrast, a taken biased or weakly biased branch may have bigger chance to be predicted as taken, so the instructions fetched after that kind of branches will be discarded and the instruction fetching will start from the target address of the taken branch on next cycle.

From the phenomenon described above, we can infer that a processor's backend execution engine will prefer threads with a more straightaway execution path.

1.3 Basic Block and Sequential Trace

Basic Block is defined as a straight-line code sequence which begins with the target of some taken branch, ends with a branch to another basic block, and with no branch in between. Basic blocks are elementary units for both instruction fetching and speculative execution.

If a basic block ends with a not-taken biased branch, then most of the time the fetch unit will continue fetch next basic block normally like the branch doesn't exist. If this kind of basic block appears successively in a thread, execution following this linear code sequence will have both good prediction rate and good instruction delivery speed. This kind of instruction stream, constructed by contiguous basic blocks with not-taken biased branches, is called "Sequential Trace" in this paper.

An overview of the relationship between basic blocks and sequential trace is illustrated in Figure 1.2. Figure 1.2(a) shows an example code of a basic block. The shadowed line is the branch that terminates the basic block. If the branch in the end of a basic block is identified as not-taken biased, like the basic block (3) and (4) in Figure 1.2(b), then the block and contiguous fall-through blocks constitute a sequential trace.

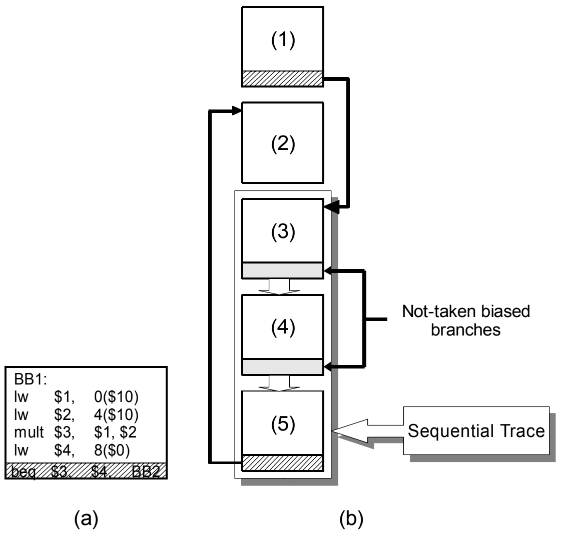

Figure 1.2: Basic Block and Sequential Trace

2. RELATED WORK

For either conventional superscalar architectures or SMT architectures, many researches have been put into the front-end architecture in an effort to improve the rate of instruction delivery to the execution core.

Seznec et al. [4] proposed a high-bandwidth design based on two-block ahead prediction. Rather than predicting the target of a branch, they predict the target of the basic block the branch will enter, which allows the critical next PC computation to be pipelined.

Tullsen et al. [2][3] not only proposed the very first model of modern SMT architectures, they also developed four fetch policies BRCOUNT, MISSCOUNT, ICOUNT, and IQPSON which improve the basic round-robin fetch policy by using feedback information from pipeline. The ICOUNT fetch policy achieves the highest performance among the four typical fetch policies and becomes the common base fetch policy in correlative papers. Their further research analyzes the effect of long-latency loads on SMT architecture and observes that freeing the resource associated with a stalled thread is better than keeping the thread ready to immediately execute upon return of the load value.

The performance of SMT architecture depends on how the fetch unit fetches instructions to fill IQs. The fetch unit must intelligently decide which threads to fetch from. Luo et al. [8] show the fetch policy that uses both fetch prioritizing and fetch gating in SMT architecture. Fetch prioritizing indicates that fetch order is decided each cycle by counting the number of unresolved low-confidence branches from threads. The threads that have the more number of unresolved low-confident branches are most likely in the wrong execution sequence. Fetch gating avoids fetching from a thread that has a stipulated number of unresolved low-confidence branches.

Kang and Gaudiot [11] build a fetch unit to control speculative execution and reduce the number of wrong-path instructions. They present a front-end mechanism, called SAFE-T, to count the number of unresolved conditional branches with low-confidence prediction in the pipeline.

El-Moursy and Albonesi [10] force on both performance improvement and power optimization. They provide three types of front-end policy based on limiting the unready instructions and data missing instructions in the queue. Their front-end policies reduce the occupancy of the instruction issue queue by increasing the number of instructions issued from nearer the head of the queue and filling the queue with instructions that are most likely to become ready for issue in the near future.

Knijnenburg et al. [9] propose a fetch policy based on dynamic branch classification mechanism to avoid fetching instructions from wrong path. The detail statement is described in section 4.1. The branch prediction accuracy is crucial for reaching the high degree of ILP in conventional architecture. Many researches about branch prediction mechanism have been proposed for increasing prediction accuracy. McFarling [1] propose the classic gshare predictor in which GHR XORs with branch address to index PHR for looking up direction of prediction. That reduces the interference between different branches with the same global history.

In [5], Chang et al. introduce a biased branch filter to reduce the interference in PHT of two-level adaptive predictors. They dynamically identify a branch as either a biased branch or not. The branch that is almost taken or almost not taken is classified as a biased branch. And a biased branch is restrained to update PHT.

Lin and Shieh [12] further apply this kind of biased branch filter combined with a confidence estimator to classify conditional branches according both their biases and their speculation confidences. This classifying information is then used to help the fetch unit in SMT processor to provide fetch gating and fetch prioritizing. Detail of this branch prediction mechanism will be described in the next section.

2.1 Introduction of Biased Branch Filter

In SMT architecture, multiple threads share processor resource to reach high hardware utilization. The branch prediction behavior may differ from the conventional superscalar architecture due to instructions from multiple threads. Moreover, the branch latency can hide by feature of SMT architecture that fetches instructions from other threads.

Lin and Shieh [12] proposed a branch prediction mechanism composed of biased branch filter, confidence estimator, and classic gshare branch predictor. This combination is claimed to achieve the follwoing goals:

1. Achieve higher branch prediction accuracy.
2. Reduce the interference of PHT.
3. Reduce competition for branch prediction mechanism.
4. Provide information for fetch unit as evidence.

The detail components of the branch prediction mechanism will be described in the following subsections.

A significant number of branches tend to be most taken or not taken that has been demonstrated in pervious researches. Some paper also showed that to separate strongly biased branches from weakly biased branches by profiling or run-time information obtain some benefit. Chang et al. [5] propose the two-level adapter branch predictors with biased branch filter. Their scheme dynamically classifies branches based on the history pattern of each branches to reduce competition for branch prediction mechanism. An identified biased branch is restrained to update the PHT. Their technique can reduce PHT interference and improve branch prediction accuracy.

In SMT architecture that multiple instructions from multiple threads run simultaneously, the difference of thread characteristic may cause some negative effect to branch predictor.

In [12], the biased branch filter consists with a 4-way associative biased table where stores the biased information of branches and a biased classifier which classifies a branch as a strongly or weakly biased branch, as shown in Fig. 2.1. The biased value in each entry is an up-down counter to indicate that the branch is taken biased, not taken biased, or non-biased. If actual direction of a branch is taken, the corresponding biased counter is increased by 1. While the biased counter reaches the taken biased threshold, the branch is classified as taken biased. If actual direction of a branch is not taken, the corresponding biased counter is decreased by 1. While the biased counter reaches the not taken biased threshold, the branch is classified as not taken biased. If the biased counter is between taken biased and not taken biased threshold, the branch is classified as weakly biased branch. While a branch is classified as a biased branch, the branch is inhibited to update PHT and biased direction is used as the prediction direction regardless of PHT.

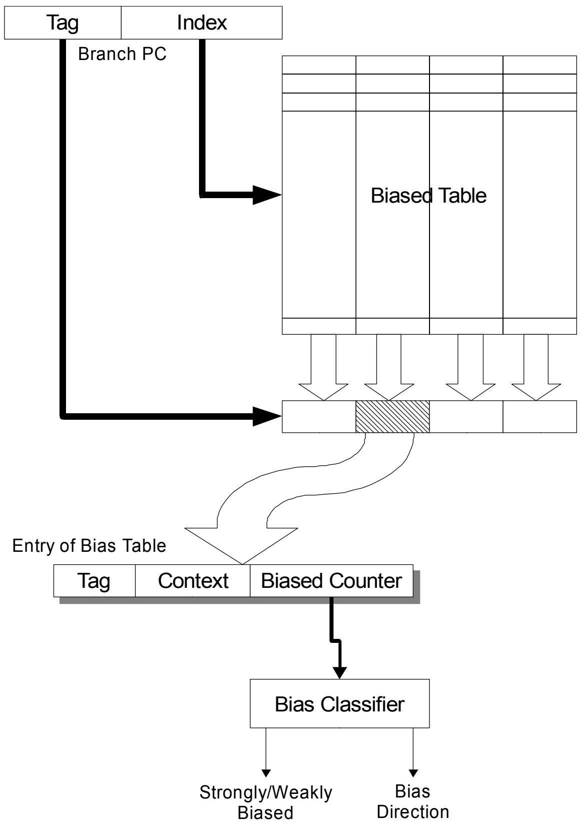

Figure 2.1: Bias Branch Filter

3. THE SEQUENTIAL TRACE MECHANISM

With the branch classification mechanism described in last chapter, we can build information about sequential traces during execution time. We designed a cache-like structure called Sequential Trace Table (STT) to record the information.In this chapter we describe detail of STT and how it works during execution time. Also we present a fetch policy using the future speculation status provided by STT.

3.1 Sequential Trace Table

Figure 3.1 shows the structure of STT. It is a cache-like 4-way set associative table indexed by the target addresses of taken branches. If a branch instruction is predicted as taken at fetch stage and there is a BTB hit, its target address will be provided by BTB for next fetch cycle. This speculative target address is also used to index into SST searching for a reference entry that may represent future speculation condition of this thread. The detail of operations on STT will be described in the following sections.

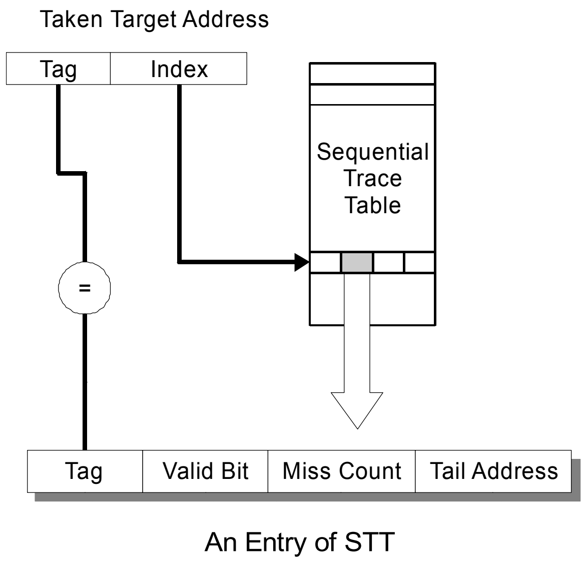

Figure 3.1: The structure of Sequential Trace Table

3.2 Operations on STT

3.2.1 Construction of STT Entry

During the commit stage of execution pipeline, if a taken branch is committed, its target address is used to update Branch Target Buffer (BTB). In our mechanism, this target address is also used to locate an entry of STT, like (1) in figure 3.2. White arrows in Figure 3.2 indicate the commitment of branches. All instructions commit in-order from left to right.

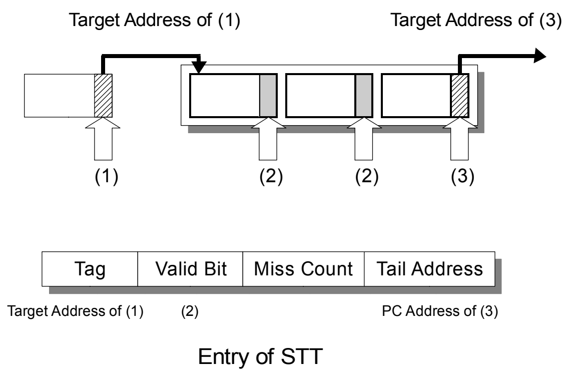

Figure 3.2: Operation during commit stage

In the following execution cycles, if another branch in the same execution route is committed as not taken, the STT entry located previously will be set as valid, like (2) in figure 3.2.

Finally there will be a taken branch called terminal branch that terminates this sequential trace, like (3) in Figure 3.2. When the terminal branch is being committed, the previously located STT entry will be checked if the valid bit has been set. Only entry with valid bit set, which means there is at least one not-taken branch in between, will be recognized as a valid entry. If the entry is valid, the address of the terminal branch will be written into the : Tail Address ; part of the STT entry. This ends the construction of an STT entry. The target address of the terminal branch will be used to locate another entry from STT, then, the step (1) to (3) in Figure 3.2 will be repeated as above.

3.2.2 Using STT for Branch Prediction

As shown in Figure 3.3, when there is no reference STT entry for the current execution route presently, the fetch engine handles branches following the normal process of branch prediction. As described in section 3.1, a branch predicted as taken will trigger a probe into STT. If there is a valid hit in the STT, the entry will be selected as a Reference STT Entry.

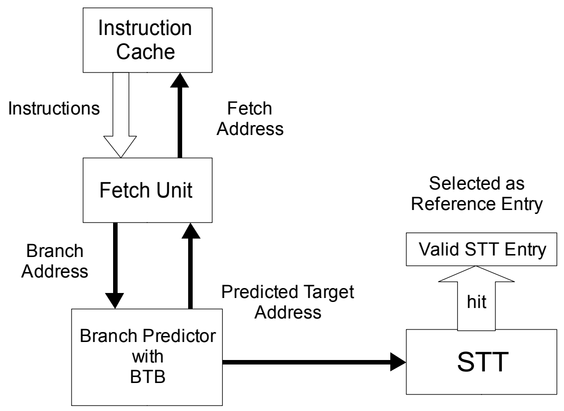

Figure 3.3: Normal Fetch process and access to STT

Figure 3.4 shows the speculation mechanism when there is a valid STT entry for current execution path. As long as the branch being fetched is inside the boundary of the sequential trace, the branch predictor will predict it as not-taken.

If the fetched branch is outside of the boundary of current sequential trace, which means the Reference STT Entry no longer represents the speculation condition of current execution route. The Reference STT Entry will be ignored, and the process of normal branch prediction and STT selection described previously will be take.

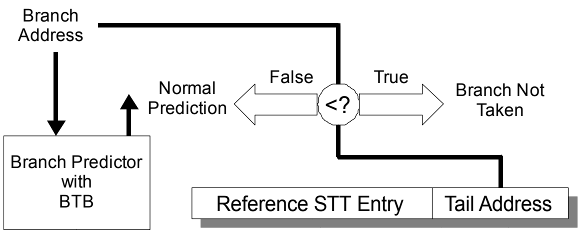

Figure 3.4: Branch Prediction using Reference STT Entry

3.2.3 Using STT for Fetch Prioritizing

Because the Sequential Trace implies a more confirmable future speculation condition, we modified the fetch algorism to give thread with longer Sequential Trace a higher fetch priority. This fetch policy totally replaced the original ICOUNT fetch policy, so the difference of instruction numbers between threads is generally not concerned about.

3.2.4 Invalidation of STT Entry

If a branch was predicted as not-take using Reference STT Entry but turned out to be a taken branch, the corresponding STT entry・s Miss Count will be increased. If the miss count reaches a certain level, the STT entry is considered as invalid. In this paper, we only tolerate 3 misses for each STT entry.

4. SIMULATION METHODOLOGY

4.1 The Simulator

The simulator used in this paper is SimpleScalar Multithreading (SSMT) originally developed by Madon et al. [7] based on the out-of-order processor model of SimpleScalar toolset [6]. It duplicates the SimpleScalar architecture's physical context according to the number of execution contexts to execute simultaneously. The SSMT simulator contains six pipeline stages: perfetch, fetch, decode, execution, writeback, and commit.

Table 4.1 shows the configuration parameters used in our simulations. We adopt ICOUNT as base fetch policy. Each cycle, at most four threads will be selected to share the fetch bandwidth.

The branch mechanism configuration is shown in Table 4.2. The extra branch misprediction penalty is set to 3 cycles for recovering the processor state, and branches are resolved after execution stage. Thus the branch misspeculation penalty is 8 cycle.

Table 4.1: Simulator parameters.

Parameter	Value
Base Fetch Policy	ICOUNT
Fetch / Issue / Commit Bandwidth	8
Fetch Queue Size	32
Register Update Unit Size	128
Load / Store Queue Size	64
Integer Add/Mult Units	8 / 2
Floating Point Add/Mult Units	8 / 2
Branch Predictor	gshare
L1 Cache Block Size	32 Byte
ICache	128KB, 2-way
DCache	128KB, 2-way
L2 Cache Block Size	64 Byte
L2 Cache	2MB, 4-way
Fast-Forward Instructions	250,000,000
Commit Instructions	50,000,000

Table 4.2: Branch mechanism configuration.

Parameter	Value
Base Branch Predictor	gshare
Pattern History Table (PHT)	2K
Global History Register (GHR)	11 bits
Branch Target Buffer (BTB)	256, 4way
Biased Table (BT)	256, 4way
Biased Counter	4bits
Taken/Not Taken Biased Threshold	12 / 3
Miss Bit Counter	4 bits
Gating Threshold	15

Table 4.3: Integer and floating point based benchmarks for simulation.

Benchmarks	
Integer Based	gzip, vpr, gcc, mcf, crafty, gap, bzip2, twolf
Floating Point Based	mesa, art, equake

Table 4.4: The selected benchmarks of each thread.

Workload	**2-Thread Benchmarks**
All Integer Based	
W21	mcf, gcc
W22	bzip2, gzip
W23	gap, twolf
All Floating Point Based	
W24	equake, art
W25	mesa, art
W26	mesa, equake
Mix of Integer and Floating Point Based	
W27	gcc, art
W28	vpr, equake
W29	bzip2, mesa
Workload	**4-Thread Benchmarks**
All Integer Based	
W41	mcf, gzip, crafty, twolf
W42	gcc, crafty, gzip, bzip2
W43	mcf, gap, bzip2, vpr
W44	mcf, crafty, gcc, vpr
Mix of Integer and Floating Point Based	

W45	mcf, bzip2, mesa, art
W46	gcc, gzip, mesa, equake
W47	twolf, vpr, mesa, art
W48	bzip2, mcf, vpr, art
Workload	**6-Thread Benchmarks**
All Integer Based	
W61	mcf, gzip, crafty, twolf, vpr, bzip2
W62	vpr, gcc, mcf, bzip2, twolf, crafty
Mix of Integer and Floating Point Based	
W63	gcc, twolf, gzip, mesa, art, equake
W64	mcf, gzip, twolf, equake, mesa, art
W65	bzip2, crafty, gzip, twolf, mesa, art
W66	mcf, gzip, crafty, twolf, gcc, art

4.2 Workloads

We selected 11 applications (alpha ISA) from the SPEC CPU2000 suite to construct our workloads where 8 of them are integer based from CINT2000 suite and the others are floating point based from CFP2000 suite. The benchmarks selected are listed in Table 4.3. All the simulations were running on a GNU/Linux x86 system with reference data sets.

Table 4.4 shows the selected combinations of 2-thread, 4-thread, and 6-thread workloads. We combine different benchmarks to form three types of workloads. These three types are integer based, floating point based and mix of both respectively.

5. SIMULATION RESULTS

In the section, we present and analyze the results of our simulation, including branch prediction accuracy and IPC comparison.

5.1 Prediction Accuracy

Figure 5.1~5.3 shows the prediction accuracy of each different fetch policy on each workload. Comparing to baseline, FB policy improve the prediction accuracy significantly because they not only reduce the occurrence of misprediction but also restrain strongly biased branches from polluting the Pattern History Table.

STT+FB shows generally the same trend as FB because the basic prediction mechanism is the same. Only when the current fetch path is inside a confirmed sequential trace, the comparison of fetch address against STT will replace the normal branch prediction algorithm. For some workloads STT outperforms FB slightly (generally less than 1%), this is because predictions of weakly-biased branches inside a sequential trace are not affected by Global History of Gshare predictor like in FB. This prediction policy is more accurate for some benchmarks with more stable sequential traces, so the performance is strongly dependent on combination of workloads.

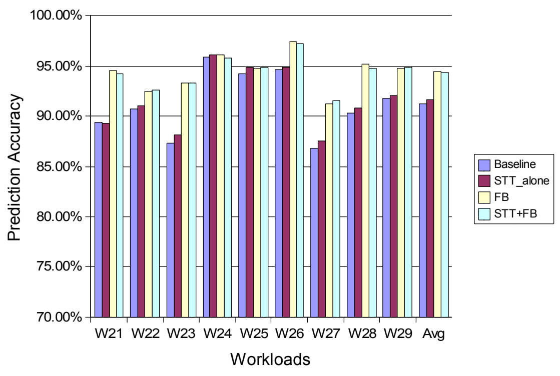

Figure 5.1: Prediction Accuracy of 2-thread workloads

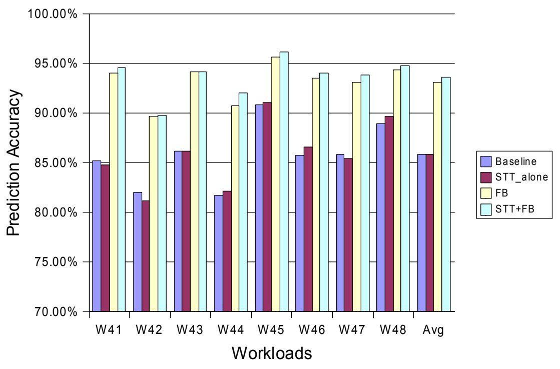

Figure 5.2: Prediction Accuracy of 4-thread workloads

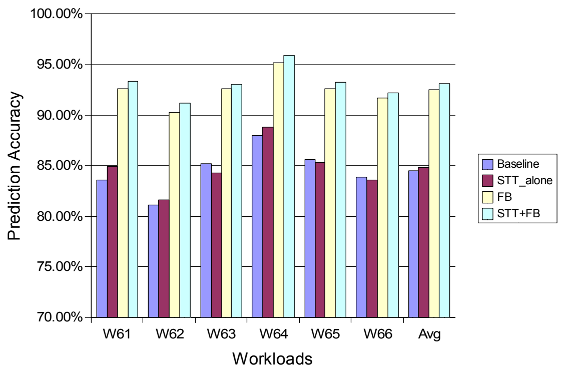

Figure 5.3: Prediction Accuracy of 6-thread workloads

5.2 Instruction Throughput

Figure 5.4 ~ 5.6 illustrate the variation of total instructions per cycle (IPC). In the simulator, the end time of threads is different that the final simulation cycle to compute IPC is unfair, so we record the termination of each thread to compute IPC for each self. Then we total IPC of threads to get the overall performance.

Figure 5.4 shows the performance of 2-thread workloads. On average, STT+FB achieves 3% gain over baseline, almost the same as the FB policy, but the differences between each workload are intense.

This is caused by the fact that our fetch policy intends to accelerate the thread having more sequential traces, by assigning more fetch slut to it, and thus clogs another. If the sequential trace information collected by STT is incorrect, a counteraction will happen.

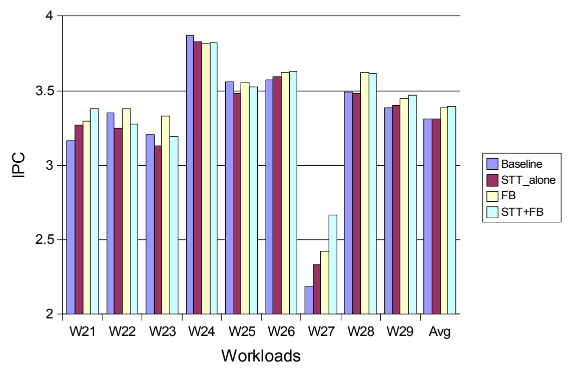

Figure 5.4: Total IPC of 2-thread workloads

Figure 5.5 and 5.6 show the results of 4-thread and 6-thread workloads. STT+FB performs an average gain of 8.9% over ICOUNT baseline and 5.6% over FB in 4-thread workloads, and achieves 15.7% and 9.8% performance gain respectively in 6-thread workloads. Again the performance of our policy fluctuates from one workload to another. One reason for this phenomenon may be that STT was indexed by the branch target address solely, which result in the inaccuracy of STT entry selection.

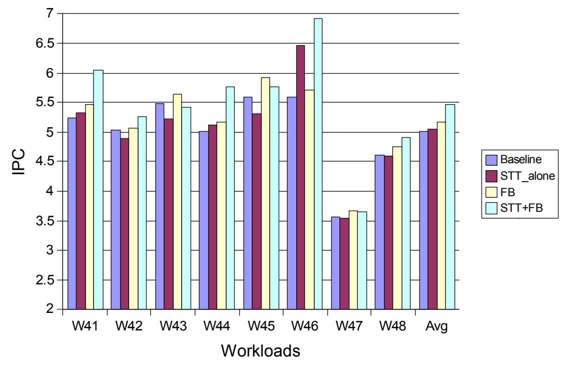

Figure 5.5: Total IPC of 4-thread workloads

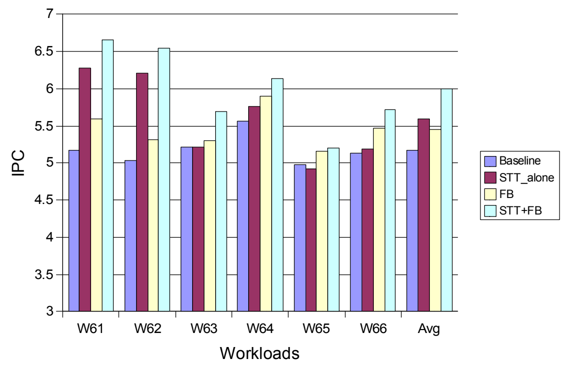

Figure 5.6: Total IPC of 6-thread workloads

6. CONCLUSIONS

One of the most important research themes for modern SMT processor development is the distribution of hardware resources across the concurrent threads. Fetch unit and branch prediction mechanism are key points of resource distribution over the whole execution pipeline, because any instruction fetched into the execution pipeline occupies processor resources no matter it is from correct or wrong execution path.

Previously proposed FB and FGAP policies use miss bit to estimate the probability of running into wrong path for each thread. It is an effective method of fetch gating but a thread can still fetch a number of instructions before the miss bit reaches the threshold.

In this paper, we propose a cache-like supplementary structure called Sequential Trace Table (STT) to provide a look-ahead into the future speculating conditions of each thread. We also propose a fetch policy to make full use of the information provided by STT. The simulation results show that the prediction accuracy is 93.7% on average compare to 93.3% by FB and 87.2% by baseline gshare predictor. Average IPC performance in 4-thread workload shows a maximal gain of 15.7% over ICOUNT baseline and 9.8% over FB, but the performance varies from one workload to another. This is a result of that our fetch policy overly depends on the benefit of sequential trace execution. A more balanced result may be expected if we can design another mechanism to make the fetch engine fall back to ICOUNT or FB policy dynamically.

REFERENCE

[1] S. McFarling, "Combiting branch predictors," Technical Report TN-36, Digital Western Research Laboratory, June 1993

[2] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," In 22nd Annul International Symposium on Computer Architecture, June 1995

[3] D.M. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor", in the Proceedings of the 23rd Annual International Symposium on Computer Architecture, May 1996

[4] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud, "Multiple-Block Ahead Branch Predictors," Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems, pp. 116-127,

Oct. 1996

[5] P.-Y. Chang, M. Evers, and Y. Patt, "Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference," Proc. Int. Conf. on Parallel Architectures and Compilation Techniques, Oct. 1996

[6] D.C. Burger and T.M. Austin, "The Simplescalar Tool Set, Version 2.0", Technical Report CS-TR-97-1342, Univ. of Wisconsin, Madison, June 1997.

[7] D. Madon, E. Sanchez, and S. Monnier, "A Study of a Simultaneous Multithreaded Architecture," In Proceedings of EuroPar'99, Toulouse, Lectures Notes in Computer Science, Volume 1685, Springer-Verlag, pages 716-726, August 31 - September 3 1999

[8] K. Luo, M. Franklin, S. Mukherjee, and A. Sezne, "Boosting SMT performance by speculation control," In 15th Proceedings of International Parallel and Distributed Processing Symposium (IPDPS), 2001.

[9] P.M.W. Knijnenburg, A. Ramirez, F. Latorre, J. Larriba, and M. Valero, "Branch classification to control instruction fetch in simultaneous multithreaded architectures," In International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'02), January 10 - 11, 2002.

[10] A. El-Moursy, and D. Albonesi, "Front-end policies for improved issue efficiency in SMT processors," 9th International Symposium on High-Performance Computer Architecture, pages 31-40, February 2003

[11] D. Kang, J.-L. Gaudiot, "Speculation Control for Simultaneous Multithreading,"Proceedings of the 18th International Parallel and Distributed Processing Symposium, Pages 76 ~ 85, April 26-30, 2004

[12] C.-H. Lin, and J.-J. Shieh, "A Study of Branch Prediction and Fetch Policy on Simultaneous Multithreading Architecture," Master thesis, Department of Computer Science and Engineering Tatung University, July, 2004