瞬間指令完成計數: 一個同步多線程的提取引擎

ICC: A Simultaneous Multithreading Fetch Engine

陳昀徽
大同大學資訊工程所
g9206011@ms2.ttu.edu.tw

謝忠健
大同大學資訊工程所
shieh@ttu.edu.tw

摘要

同步多線程(SMT)是一種允許在每一個週期能夠同時發派來自不同獨立的應用程式或是線程的指令的一種技術。提取單元一直被認為是同步多線程的主要瓶頸所在，過去許多研究曾提出過一些提取策略來增進提取效率以及整體的效能。

在此篇論文，我們提出一個全新的提取策略，稱之為瞬間指令完成計數(ICC)，它會計算每個線程在每一個時脈確認完成的指令數目，然後依照這些資訊來決定下一個週期要從哪些線程來提取指令。此外，我們還將此提取策略和被稱之為提取偏向(FB)和提取閘控優選(FGAP)的分支機制做結合，來建構更有效率的提取單元。經由模擬結果顯示，整體效能提升大約百分之十三，並且還減少了發派佇列的使用大小，同時還減少錯誤路徑指令的提取。另外，我們還展示負載平衡的狀態，這是過去相關研究沒有詳細討論過的議題。

關鍵字： 同步多線程、提取策略、提取單元。

Abstract

Simultaneous Multithreading (SMT) is a technique that permits multiple instructions from multiple independent applications or threads to issue each cycle. While the fetch unit has been identified as one of the major bottlenecks of SMT architecture, several fetch schemes were proposed by prior works to enhance the fetching efficiency and overall performance.

In this paper, we propose a novel fetch policy called Instantaneous Commit Count (ICC) which counts each thread's retired instructions each cycle then properly selects which threads to feed next cycle. We also combine this scheme with branch mechanisms, named FB and FGAP, to construct the effective fetch unit. Simulation results show that the overall performance is improved about 13% on speedup, the issue queue size is reduced and the wrong-path instructions fetch are also reduced. Furthermore, we show the state of load balance that never discussed in prior works in detail.

Key words：Simultaneous Multithreading (SMT), Fetch Policy, Fetch Unit

1. INTRODUCTION

1.1 Simultaneous Multithreading Architecture

Simultaneous Multithreading (SMT) [1,2,3] is a technique that permits multiple instructions from multiple independent applications or threads to issue each cycle. All threads in an SMT processor are active simultaneously, competing each cycle for all available resources. This dynamic sharing of the functional units allows SMT to substantially increase throughput by hiding most per-thread latency. SMT also achieves three goals: (1) minimizes the architectural impact on the conventional superscalar design, (2) has minimal performance impact on a single thread executing alone, and (3) achieves significant throughput gains when running multiple threads. SMT architecture is a straightforward extension to the conventional superscalar design. Thus, nearly all hardware resources remain completely available even when there is only a single thread in the system. The changes necessary to support SMT are as follows [2]:

1. Multiple program counters and fetch unit has ability to fetch instructions from different threads each cycle.
2. Private return address stacks for each thread to provide subroutine return destinations.
3. Per-thread instruction retirement, instruction queue flush, and trap mechanisms.
4. The branch target buffer entry adds a thread-id field to avoid predicting phantom branches.
5. A large register file supports logical

registers of all threads plus additional registers for register renaming.

Figure 1.1 [3] illustrates the difference between superscalar, fine-grained multithreading, and SMT by showing sample execution sequences of the three architectures. Each row represents the slots issued per cycle. Assuming four instructions in maximum can be issued each cycle. An empty box indicates that there is no instruction chosen to fill this issue slot; respectively, a filled box represents that slot has been fed by an instruction from a thread. Two types of waste are identified in the picture. Horizontal waste occurs when some of the issue slots in a cycle can not be used. It typically means poor instruction-level parallelism. Vertical waste occurs if all slots were not used in a cycle. It occurs when a long latency instruction that prevents further instructions from issuing.

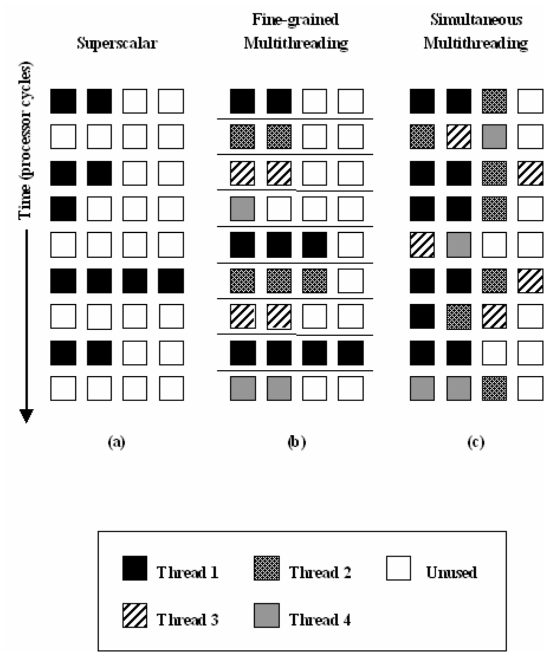

Figure 1.1: The comparison of issue slot between three architectures.

Figure 1.1a shows the sequence of a conventional superscalar. Superscalar processors fetch multiple instructions and issue them from a single program or thread. When it cannot find any instructions to issue in a cycle, both horizontal and vertical wastes will occur. Fine-grained multithreaded processors maintain thread states and quickly switch between them every cycle, that is, they execute instructions from a thread at one cycle and switch to another thread at the next cycle. As the Figure 1.1b shows, it can hide the long-latency operations and eliminating vertical waste, but horizontal waste still exists. As horizontal waste can not be removed, multithreaded architectures will be limited by the instruction-level parallelism in a single thread as superscalar processors while instruction issue width continues to increase.

SMT tries to conquer both horizontal and vertical wastes by allowing instructions from multiple threads to execute in a single cycle as shown in Figure 1.1c. Because it selects instructions from several threads, instruction-level parallelisms from all threads are exploited, eliminating horizontal waste. And if one thread is blocked due to long-latency instructions or resources conflicts, unblocked threads can use these slots, thus vertical waste is also eliminated.

1.2 Bottlenecks of SMT

Although the SMT architecture dynamically sharing the processor resources to exploit both the thread-level parallelism came from multiple threads and instruction-level parallelism from single thread and better utilizing the resources, there are several bottlenecks identified [2].

SMT improves performance in the benefits of dynamic sharing of resources, but it does appear to have some potential drawbacks due to inter-thread contention. Instructions competed for resources now coming from multiple threads instead of single thread puts greater stress to the shared structures such as caches, translation look-aside buffers and branch target buffers than traditional processors do. For example, sharing the cache with multiple threads, that is, partitioning the cache into pieces for threads will eventually reducing the cache space used by each thread, hence decrease the degree of locality and cause cache misses to arise. Instruction fetching unit is one of the major performance bottlenecks which also widely studied [2,7,8,9,10,11,12]. On one hand, the SMT fetch unit benefits from inter-thread competition for instruction bandwidth by partitioning the bandwidth among threads and finding more useful instructions to fill the issue slot, which is often difficult to fill if there is only one thread to be accessed at a time. On the other hand, dynamic scheduler of SMT processors which issuing more instructions (from multiple threads) than traditional processors (from a single thread) does put more stress on fetch unit. It must now fetch more instructions to keep pace with the speed that consumed by later pipe-stages. In order to improve fetch efficiency, the fetch unit must smart enough to determine which thread to fetch from since there may be several threads running at a given time. Several fetch schemes have been proposed to improve the SMT performance [2,7,8,9,10,11,12].

Another problem is the impact of the long-latency instructions. This happens when the memory-bond threads or threads with high concentration of long-latency instructions fills the instruction scheduling window with instructions that cannot be issued quickly hence prevent other threads to be fetched and even worse, stall the processor. This problem can be solved by either increasing the size of instruction queue or good fetch scheme design.

In this paper, we propose a novel fetch scheme to overcome the foregoing bottleneck. We attempt to let properly threads whose flowing speed is better to get more resources so these threads won't suffer from the other slow-flowed threads. We will describe our policy with detail in the section 3.

1.3 The Paper Organization

This paper is organized as follows. In section 2, we review related works. We describe baseline fetch policy and present our fetch policy on SMT in section 3. Then, section 4 and 5 shows the methodology and analyzes the simulation results. Finally, section 6 concludes the paper.

2. RELATED WORKS

Tullsen et al. [1,3] proposed the SMT architecture and firstly implemented it on MIPS R10000 and DEC Alpha platform. The SMT architecture doesn't heavily impact on the conventional superscalar design. They also studied fetch policies for SMT processors and investigated several fetch policies in [2], such as ICOUNT, BRCOUNT, IQPOSN and MISSCOUNT which attempt to improve on the simple round-robin priority policy by using feedback from the processor pipeline. With their experiments, the ICOUNT has the best performance. In particular, the ICOUNT fetch policy has been chosen by many researches as their base fetch policy. They describe another problem that is when a single thread with poor cache performance can strangle overall SMT performance, by monopolizing resources that could be exploited by other threads. The solution proposed in [5] is to free resources occupied by a stalled thread by flushing its instructions from the pipeline.

The Intel Pentium 4 processor [6] is the first commercially available general-purpose processor to implement a simultaneously multithreading core. The Pentium 4 supports a dual-threaded implementation of SMT called Hyper-Threading Technology. Hyper-Threading Technology makes a single physical processor appear as dual logical processors. The primary difference between the Hyper-Threading implementation and the architecture proposed in the SMT research is the mode of sharing of hardware structures. While the SMT research indicates that virtually all structures are more efficient when shared dynamically rather than partitioned statically, some structures in the Hyper-Threading implementation such as the ROB entries and load/store buffers are statically divided in half when both threads are active.

C. Shin and S. Lee [7] has investigated how much more improvement can be made by allowing an adaptive dynamic thread scheduling approach rather than the fixed scheduling approaches employed in earlier work [2]. He proposed the detector thread approach to implement adaptive scheduling (when to choose from ICOUNT, BRCOUNT and MISSCOUNT) with low hardware and software overhead. The detector thread is a special thread that occupies one designated thread context with minimal extra hardware. It is scheduled for execution when idle slots are available.

A. El-Moursy and D. Albonesi [8] proposed several fetch policies that reduce the requirement of integer and floating point issue queue sizes in SMT processors. Their schemes are based on ICOUNT policy, and provide some gating mechanisms on the basic policy according to the number of the instructions that are not ready in the issue queue, or the L1 cache miss rate of a given thread. For the same level of performance, it achieves 33% reduction in the occupancy of the issue queue.

A. Falcon et al. [9] have shown that implementing a fetch architecture fetching from more than one thread is too expensive, both in terms of cost and complexity. They have demonstrated that a solution to increment the SMT fetch performance is not to fetch few instructions from several threads, but to fetch many instructions from a single thread. They use a technique called stream fetch which allows the fetch unit to fetch a basic block between two predicted taken branches. Implementing a fetch unit fetching only from one thread solves the problem of the high complexity of the SMT fetch unit.

E. Fernandez et al. [10] proposed a fetch policy, which called DWarn. DWarn uses L1 misses as indicators of L2 misses, giving higher priority to threads with no outstanding L1 misses. DWarn acts on L1 misses, before L2 misses happen in a controlled manner to reduce resources underuse and to avoid harming a thread when L1 misses do not lead to L2 misses. Their results show that DWarn outperforms previously proposed policies [2], in both throughput and fairness, while requiring fewer resources and avoiding instruction re-execution.

L. He and Z. Liu [11] also proposed a fetch scheme that constructs a formula to calculate the needed number of instructions to every selected thread. Although the results have showed that it can improve speedup and reduce the size of issue queue, the formula only focused on 2-threads.

Yang and Shieh [12] proposed a dynamical fetch scheme which gives the highest fetch priority to the long latency bound threads while the RUU and LSQ is under low usage. Their motivation is to gain further performance by not only use the resources effectively but also by the urgency of the instructions.

In the following sections, we will describe base fetch policy and propose a new effective fetch policy. We will combine the fetch scheme with newly proposed branch prediction mechanism [13]. Finally, implementation technique and simulation results will be analyzed.

3. FETCH POLICY ON SMT

3.1 Base Fetch Scheme

In order to have high performance on SMT, the fetch unit must get as many instructions as possible each cycle. Fetching from a single thread seems to be insufficient to feed an 8-way execution core. Solutions that try to widen the fetch throughput are targeted to fetch from multiple threads in a single cycle. A priority policy is used to decide which thread should be fetched first. In this paper, the fetch unit can fetch four instructions per thread from two threads each cycle. By the way, the fetch unit should fetch from two available threads which have higher priority.

Tullsen et al. [2] have proposed and studied several fetch policies for SMT. Their results show that the best priority policy is ICOUNT, in which priority is assigned to a thread according to the number of instructions it has in the decode, rename, and issue stages (issue queues) of the pipeline. Threads with the fewest such instructions are given the highest priority for fetch. The rationale is that such threads may be marking more forward process than others. It can prevent one thread from clogging the issue queue, and provide a mix of threads in the issue queue to increase parallelism. In this paper, we take ICOUNT as our baseline fetch policy.

3.2 Our Proposed Fetch Policy

We found that ICOUNT fetch policy overly attempts to equally divide the occupancy of the issue queue with each thread. For example, ICOUNT policy continuously fetches the thread which just squashed because of its mis-speculative behavior. Furthermore, ICOUNT scheme causes issue queue full by seriously competing when we get more available threads. So, if we can properly let threads whose flowing speed is better to get more resources, these threads won't suffer from the other slow-flowed threads. Basing on this idea, the fetch unit properly response to share all resources on every situation, for example, outstanding cache miss. In this paper, we propose a novel fetch policy called Instantaneous Commit Count (ICC) which counts each thread's retired instructions each cycle then selects which threads to feed next cycle.

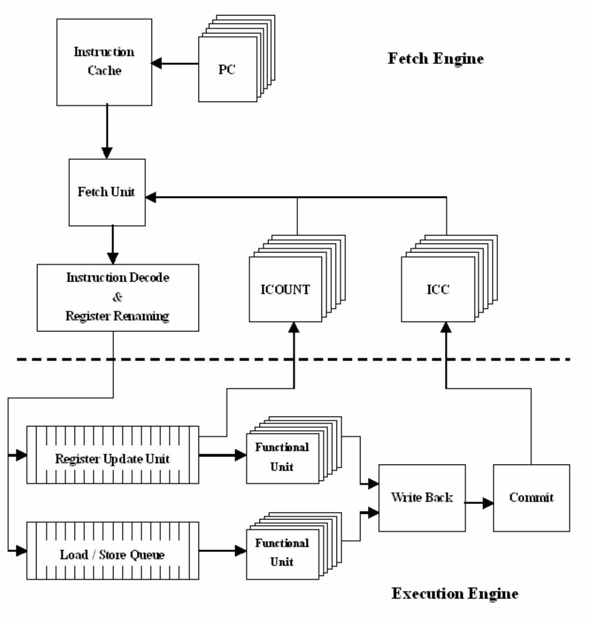

Figure 3.1: The architecture of ICOUNT and ICC.

To implement the scheme, as in ICOUNT, it needs a counter for each thread to record its retired instructions number every cycle. Figure 3.1 shows the architecture of ICOUNT and ICC fetch policy. Here, we create three types of ICC fetch policy as follow:

1. ICC-1: The fetch unit always prefers to fetch the threads which outperform over the others at single cycle. That means we will select the threads whose ICC is higher. This scheme leads the threads which have better throughput at single execution to early complete their job, but it causes bad load balance. We will discuss this problem in section 5.
2. ICC-2: In order to avoid favoring excessively some threads, we should adopt ICC-1 at appropriate time. While issue queue is under low usage, we let the threads which have lower ICC value to take higher fetching priority. Otherwise, we choose ICC-1 to be the fetch policy. To do this, it will get better load balance than ICC-1 policy.
3. ICC-3: Here, we keep all threads at least N instructions in issue queue. If all threads exceed N instructions in issue queue, we will use ICC-1 fetch policy. With our experiment, we take N = 6 as it shows both the best average performance and load balance. In this scheme, when some thread has been squashed, it will get at least two chances to fetch during the next two

cycles. Therefore, with this policy, it can properly and effectively share the rest of issue queue unlike ICOUNT tend to equally divide whole issue queue.

Our goal is to allocate more reasonably the resources of SMT architecture than base fetch scheme. If we can increase the usable issue queue entry of specific threads at the right moment, the throughput should enhance apparently.

3.3 Dynamically Speculative Controlled Fetch Policy

We have proposed the branch prediction mechanism with biased branch filter and confidence estimator to reduce the competition for branch predictor between thread and classify conditional branches as biased or confident branches in [13]. Therefore, the fetch unit that plays an important role in the SMT architecture decides which threads to fetch instructions from each cycle according to the information from the proposed branch prediction mechanism.

Besides, we also introduced extra counter, called *miss bit* (4 bits), for reordering the fetch priority that is based on ICOUNT fetch policy originally. The *miss bit* counters are used to estimate the confidence of threads and each thread has its own *miss bit* counter. A thread with larger *miss bit* value is considered to more likely fetch instructions from wrong path. Here, we combine this mechanism with our new strategy for fetch instructions. We just choose ICC-3 fetch policy because of its better load balance and performance after combining.

ICC-3 + FB (Fetch Bias): The branch prediction mechanism is *gshare* predictor with the biased branch filter which uses a biased counter to determine the bias of branch. While a fetched branch is classified as a strongly biased branch, the *miss bit* counter of corresponding thread decreases by 1. While a weakly biased branch is fetched, the *miss bit* counter increase by 8. If *miss bit* counter is greater than the gating threshold of 15, the thread is stalled to fetch. The fetch priority is constructed by ICC-3 and reconstructed by sorting the *miss bit* counter.

ICC-3 + FGAP (Fetch Gating and Prioritizing): The branch prediction mechanism is *gshare* predictor with the integration of biased branch filter and confidence estimator. The fetch policy combined with fetch gating and fetch prioritizing is shown in Figure 3.2. How the *miss bit* counter and ICC counter attain to fetch gating and prioritizing is described as follows:

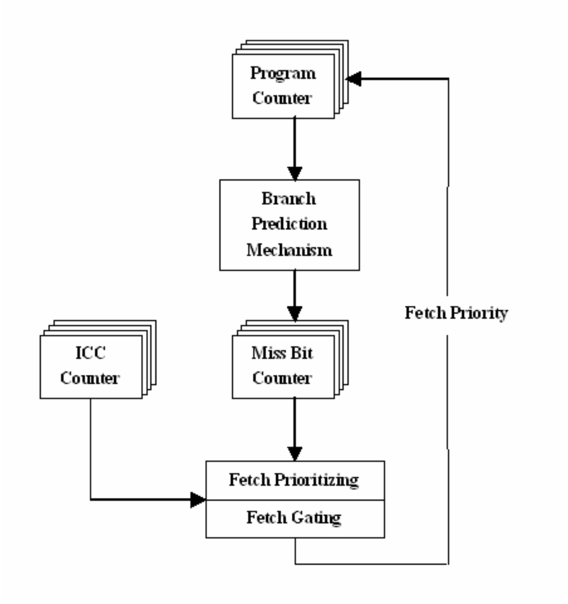

Figure 3.2: The schematic diagram of the combined fetch policy.

1. The fetch unit fetches instructions from instruction cache according to fetch priority that decided in prior cycle.
2. While a branch instruction is fetched (hit on BTB), program counter of the branch is delivered to branch predication mechanism to obtain the prediction result and classification information.
3. If the branch is classified as a strongly biased or high-confident branch and hit in BTB, the *miss bit* counter of the corresponding thread is decreased by 1. If the branch is classified as low-confident branch, the *miss bit* of the corresponding thread is increased by 8. If the branch is classified as non-confident branch or misses in BTB, the *miss bit* of the corresponding thread is set to 15.
4. The fetch unit sorts ICC counter and looks up all *miss bit* counter to decide the fetch priority to be used in next cycle. If the *miss bit* counter of first fetch priority thread is greater than gating threshold (default 15), the fetch priority is reset to give the highest priority to the thread with lowest *miss bit* value.

However, if there are not enough instructions in the IQs, the fetch unit continues to fetch instructions regardless of *miss bit* counter. The fetch priority is constructed by ICC-3 and reconstructed by sorting the *miss bit* counter.

4. METHODOLOGY

Our simulator is derived from the SimpleScalar Multithreading (SSMT) simulator which originally developed by Madon et al. [4]. The simulator implements simultaneous multithreaded processor pipeline based on the out-of-order processor model from SimpleScalar tool set [16]. It duplicated the SimpleScalar architecture's physical context according to the number of execution contexts to execute simultaneously.

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 Functional Units	8
Floating Point Units	8
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: Instruction and memory access latency.

Latency Type	Cycles
Integer	1
FP Add	2
FP Multi	4
FP Div	12
L1 Cache Hit	1
L2 Cache Hit	10
Memory	80~122

Table 4.3: 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
Confidence Counter	4 bits
Confidence Threshold	7
Non-Confidence Threshold	2
Miss Bit Counter	4 bits
Gating Threshold	15

Table 4.4: 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 4.5: The selected benchmarks of each thread.

Workload	2-Thread Benchmarks
All Integer Based	
1	gzip, bzip2
2	gap, twolf
All Floating Point Based	
3	mesa, art
4	mesa, equake
Mix of Integer and Floating Point Based	
5	vpr, equake
6	bzip2, mesa
Workload	**4-Thread Benchmarks**
All Integer Based	
1	mcf, gzip, crafty, twolf
2	mcf, gap, bzip2, vpr
3	mcf, crafty, gcc, vpr
Mix of Integer and Floating Point Based	
4	mcf, bzip2, mesa, art
5	gcc, gzip, mesa, equake
6	gcc, crafty, gzip, mesa
Workload	**6-Thread Benchmarks**
All Integer Based	
1	mcf, gzip, crafty, twolf, vpr, bzip2
2	vpr, gcc, mcf, bzip2, twolf, crafty
Mix of Integer and Floating Point Based	
3	gcc, twolf, gzip, mesa, art, equake
4	mcf, gzip, twolf, equake, mesa, art
5	mcf, gzip, crafty, twolf, gcc, art
Workload	**8-Thread Benchmarks**
Mix of Integer and Floating Point Based	
1	mcf, gcc, gzip, crafty, twolf, bzip2, vpr, art
2	mcf, gcc, gzip, gap, bzip2, vpr, art, mesa
3	mcf, gcc, gzip, twolf, bzip2, mesa, art, equake
4	mcf, vpr, gzip, twolf, bzip2, mesa, art, equake

Table 4.1 describes our configuration of parameters on SSMT. We adopt ICOUNT as our base fetch policy to compare with our proposed scheme. In this paper, the fetch unit can fetch four instructions per thread from two threads each cycle. Table 4.2 shows execution time of instructions and memory access latencies.

The branch mechanism configuration is shown in Table 4.3. The extra branch mis-penalty is set to 3 cycles for recovering the processor state and branches are resolved after execution stage.

Thus the branch mis-speculation penalty is 8 cycle.

We picked up 11 applications (alpha ISA) from the SPEC CPU2000 suite to construct our workloads where 8 of them were integer based from CINT2000 suite and others were floating point based from CFP2000 suite. The benchmarks selected are listed in Table 4.4. All the benchmarks were running on a GNU/Linux x86 system using reference data sets.

Table 4.5 shows selected workloads of threads (2, 4, 6 and 8 threads). 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 this section, we present the simulation results of our experiment. We will show the speedup of our fetch policy over ICOUNT scheme. Then we discuss the influence of our policy and ICOUNT scheme on the issue queue utilization, wrong-path fetch rate and load balance.

5.1 Experiment Results

Figure 5.1 to 5.4 show the number of instructions fetch per cycle normalized to base fetch scheme. As we can see, our schemes achieve same value (the difference within 2% on average) in 2-thread workloads because the fetch unit can only fetch from two threads each cycle. The reason less fetch rate in workload 1 (gzip, bzip2) is the fetch gating mechanism often stall threads. In 4-thread workloads, ICC-1, ICC-2, ICC-3, ICC-3+FB and ICC-3+FGAP get 5.7%, 1.6%, 3%, 4.2% and 2.4% improvement on average respectively.

Here, in workload 6 (gcc, crafty, gzip and mesa) our fetch policy get fewer fetch numbers because it may get worse mix of instructions and result in more long data dependency chain. So, our fetch policy which combines with FB and FGAP will lead to fetch less instructions. In 6-thread workloads, ICC-1, ICC-2, ICC-3, ICC-3+FB and ICC-3+FGAP enhance 18.8%, 8.2%, 6.1%, 4.4% and 1.6% on average respectively. Finally in 8-thread workloads, ICC-1, ICC-2 and ICC-3 achieve 25.1%, 13.7% and 7.3% improvement on average respectively. Both FB and FGAP fetch gating mechanism lead to reduce wrong-path instructions fetching. Therefore, the speedup of instructions fetch per cycle for these two schemes is unapparent. Overall, fetch policy with fetch gating mechanism, FB and FGAP, will cause fewer fetch instructions because of wrong-path fetch reducing. Our ICC scheme can get higher fetch number with more threads (6 threads or 8 threads).

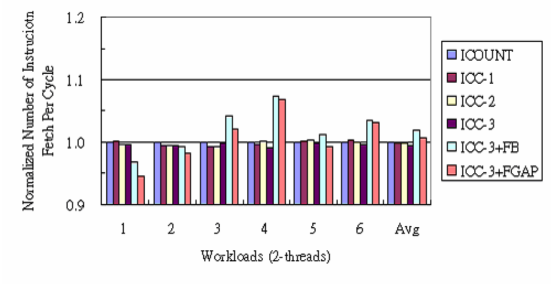

Figure 5.1: Normalized fetch rate of 2-threads workload.

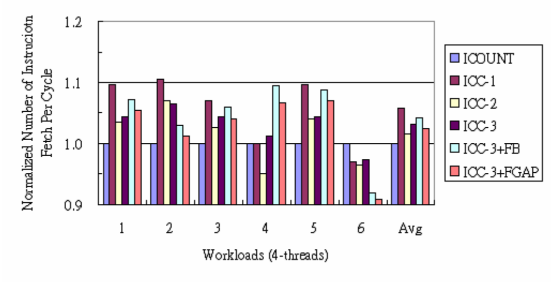

Figure 5.2: Normalized fetch rate of 4-threads workload.

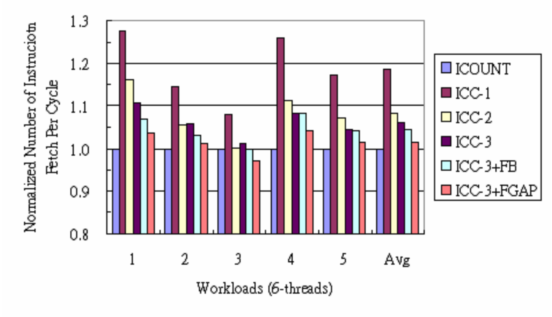

Figure 5.3: Normalized fetch rate of 6-threads workload.

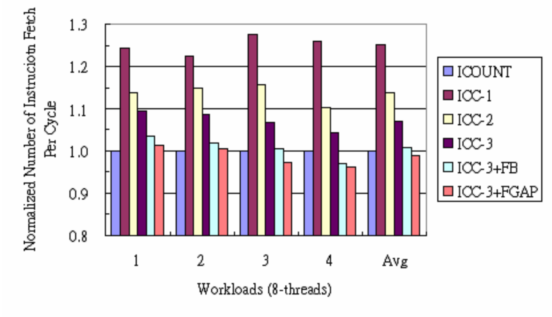

Figure 5.4: Normalized fetch rate of 8-threads workload.

The IPCs of the combined workloads are shown in Figure 5.5 to 5.8. Our commit counter just record instantaneous IPC for each threads. As illustrating, 2-thread workloads get almost same performance on each scheme except fetch policy with FB and FGAP which increase 8.2% and 8.9% speedup on average. In 4-thread workload, ICC-1, ICC-3, ICC-3+FB and ICC-3+FGAP achieve 3%, 2.6%, 11.6% and 13% performance improvement on average respectively. Although ICC-2 gets lower IPC, the gap is within 1%.

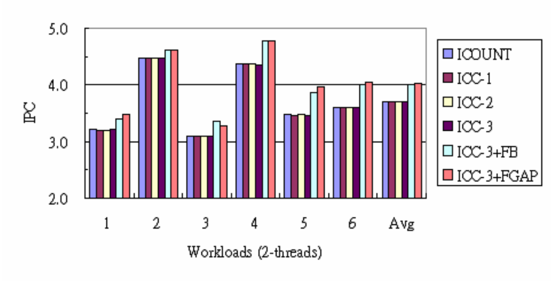

Figure 5.5: Performance of 2-thread workloads.

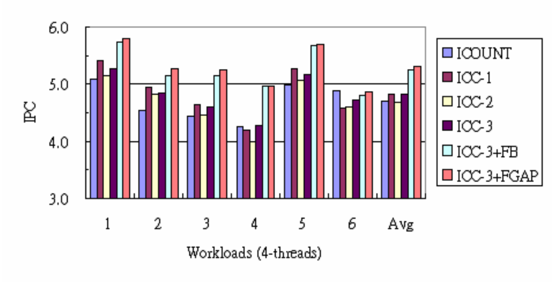

Figure 5.6: Performance of 4-thread workloads.

In 6-thread workloads, ICC-1, ICC-2, ICC-3, ICC-3+FB and ICC-3+FGAP enhance 14.6%, 5%, 4.8%, 11.9% and 12.6% performance on average respectively. Finally in 8-thread workloads, ICC-1, ICC-2, ICC-3, ICC-3+FB and ICC-3+FGAP obtain 21.7%, 9.9%, 6.8%, 9.4% and 11.5% throughput improvement on average respectively. ICC-1 extremely favor the thread whose flowing speed is better so the first finished and second finished thread will lead baseline so much and gain obviously performance. Nevertheless, ICC-1 results in bad load balance because it makes the finished distance between slow-flowed thread and fast-flowed thread more enormous. We will discuss this problem at next subsection in detail.

The IPCs of 6-thread workloads is higher than that of the 8-thread workloads for ICOUNT policy. Two facts may explain this phenomenon. First fact is that all 8-thread workloads are all mix type but several 6-thread workloads are all integer type. Another and most important fact is that the resources are severely and overly competed when fetch unit get 8 threads to choose.

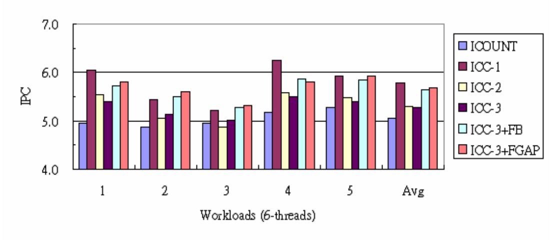

Figure 5.7: Performance of 6-thread workloads.

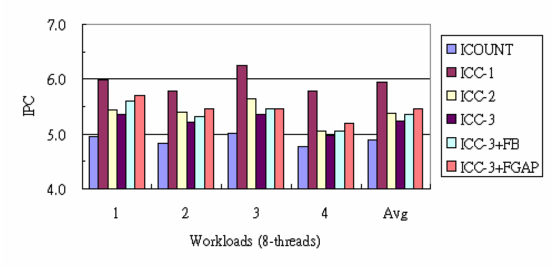

Figure 5.8: Performance of 8-thread workloads.

Figure 5.9 shows average issue queue occupancy of each fetch scheme. It is obvious that ICOUNT's usage of issue queue is higher than our policy when getting more available threads (6 or 8 threads). When upping to 8-threads, ICOUNT scheme even occupy 112 entries of issue queue on average. ICOUNT scheme often fills up issue queue to stall pipeline processing. Here, ICC-1, ICC-2, ICC-3, ICC-3+FB and ICC-3+FGAP reduce 15.1%, 11%, 5.9%, 7% and 17.4% occupancy of issue queue on 8-thread workloads respectively.

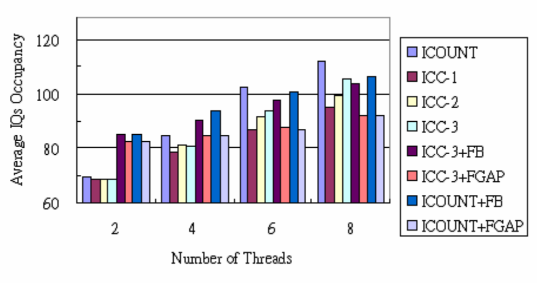

Figure 5.9: Average occupancy of issue queue.

Figure 5.10 and Figure 5.11 illustrate the percentage of wrong-path fetch and execution for each policy. ICC-3+FB and ICC-3+FGAP reduce 36.4% and 54.9% wrong-path fetch and 27% and 45.9% wrong-path execution respectively in 8-thread workloads.

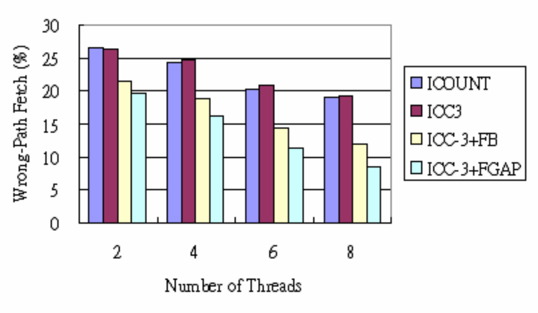

Figure 5.10: Percentage of wrong-path fetch.

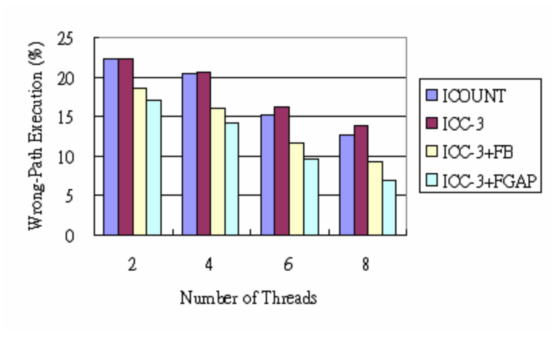

Figure 5.11: Percentage of wrong-path execution.

5.2 Load Balance on SMT

In this section, we discuss the load balance on SMT architecture. Although there are many researches on fetch policy [7, 8, 9, 10, 11, 12, 13], they never present or describe their load balanced states in detail. In a general purpose computer, most users would like to quickly finish each work on average. Basing on this point, ICOUNT scheme provides pretty nice load balance because it attempts to keep equally utilization of issue queue. However, our fetch policy tries to favor a thread which has a fast flow speed. In other words, our policy attempts to finish one thread as soon as possible and to maintain the load balance like ICOUNT.

Figure 5.12 shows the load balance of all policy for each thread. We define "best load balance" should be a horizontal line whose slope is zero. When the slope gets higher, the work gets worse load balance. Even so, better load balance does not mean higher performance. Here, ICC-1 enhances performance outstandingly but it gets worst load balance.

As we can see, ICC-1 let "fast thread", which achieves higher IPC than others in single threaded mode, to finish more early but it also let "slow thread", which achieves lower IPC than others in single threaded mode, to finish more lately. Therefore, we attempt to resolve this situation and then propose ICC-2 scheme which let the threads that have lower ICC value to take higher fetching priority while issue queue is under low usage. Although, ICC-2 tries to avoid overly favoring specific threads, it doesn't seem to work effectively. Finally, we present the better resolution, ICC-3 fetch policy, which keeps all threads at least N instructions in issue queue. If all threads exceed N instructions in issue queue, we will use ICC-1 fetch policy. ICC-3 policy achieves our goal to not only improve performance but also maintain load balance like ICOUNT scheme. As illustrated, ICC-3 policy can let "fast thread" to finish as early as possible but it doesn't delay "slow thread". So, we combine our previously proposed branch mechanism with ICC-3 policy cause its better behavior.

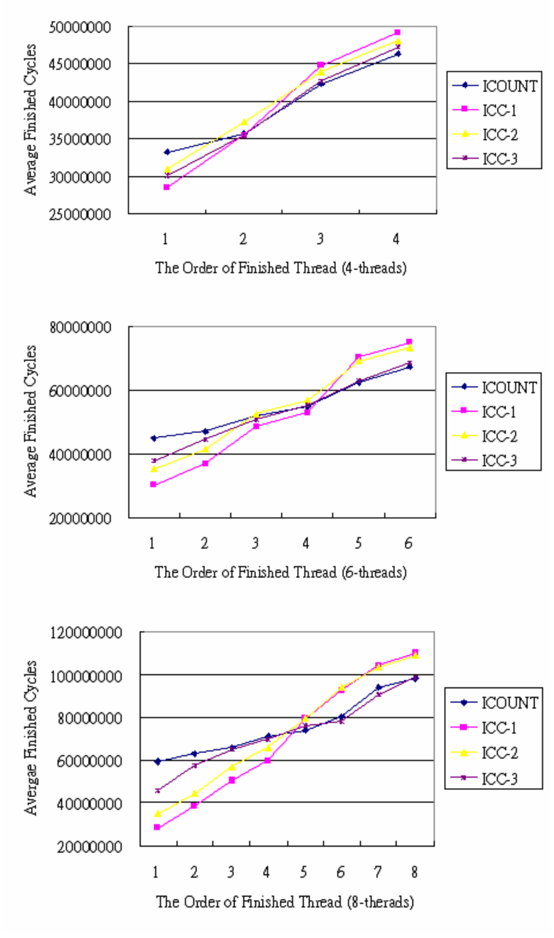

Figure 5.12: The order of finished thread and average finished cycles.

6. CONCLUSIONS

When the SMT processors gain performance by sharing the processor resources dynamically to exploit both thread-level parallelism and instruction-level parallelism, it still has some potential drawbacks. The fetch unit has been identified as one of the major bottlenecks of SMT architecture. Several fetch schemes were proposed by prior works to enhance the fetching efficiency. Among these schemes, ICOUNT, proposed by Tullsen et al. in which priority is assigned to a thread according to the number of instructions it has in the decode unit, register renaming unit and instruction queues were considered to be a great scheme not only the performance and load balance but also the efficiency of implementation.

We found that ICOUNT fetch policy overly attempts to equally divide the occupancy of the issue queue with each thread. When one thread squashed because of mis-speculation, ICOUNT policy rushes to fill this thread until approximately same utilization of issue queue. To allocate resources more reasonably, we propose a novel fetch scheme called Instantaneously Commit Count (ICC) which counts each thread's retired instructions each cycle then selects which threads to feed next cycle. Moreover, we divide ICC policy into three types to simulate. Although ICC-1 can achieve the highest performance speedup up to 21.7%, but it has worst load balance. Also, ICC-2 tries to improve this phenomenon but failed. Finally, we find ICC-3 not only enhance performance but also keep good load balance like ICOUNT.

We also proposed branch mechanism to combine with ICC-3 scheme because of its good effect on both throughput and load balance. With our experiment, ICC-3+FB and ICC-3+FGAP increase performance speedup up to 11% and 12.4% respectively over baseline. ICC-3+FB and ICC-3+FGAP increase performance speedup up to 6.2% and 7.5% respectively over ICC-3. Furthermore, our policy also reduces the average issue queue size.

REFERENCE

[1] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," *In 22nd Annul International Symposium on Computer Architecture*, June 1995, Pages 392-403

[2] D. 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 23rd Annul International Symposium on Computer Architecture*, May 1996

[3] S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm, and D. Tullsen, "Simultaneous multithreading: A platform for next-generation processors," *IEEE Micro*, Sep. 1997, Pages 12-18

[4] 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, Sep. 1999, Pages 716-726

[5] D. Tullsen and J. Brown, "Handling Long-latency Loads in a Simultaneous Multithreading Processor" *MICRO-34*, Dec. 2001, Pages 318-327

[6] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller and M. Upton, "Hyper-Threading Technology Architecture and Microarchitecture" *Intel Technology Journal Q1*, 2002

[7] C. Shin and S. Lee, "Dynamic Scheduling Issues in SMT Architectures," *In Proceedings of the 17th International Parallel and Distributed Processing Symposium*, April 2003, Pages 8pp.

[8] A. El-Moursy and D. Albonesi, "Front-end policies for improved issue efficiency in SMT processors," *In Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, Feb. 2003, Pages 31-40

[9] A. Falcon, A. Ramirez and M. Valero, "A Low-Complexity, High-Performance Fetch Unit for Simultaneous Multithreading Processors," *In Proceedings of the 10th International Symposium on High Performance Computer Architecture*, Feb. 2004, Pages 244-254

[10] E. Fernandez, F. Cazorla, A. Ramirez and M. Valero, "DCache Warn: an I-Fetch Policy to Increase SMT Efficiency," *In Proceedings of the 18th International Parallel and Distributed Processing Symposium*, April 2004, Pages 74-84

[11] L. He and Z. Liu, "An Effective Instruction Fetch Policy for Simultaneous Multithreaded Processors," *In Proceedings of the 7th International Conference on High Performance Computing and Grid in Asia Pacific Region*, July 2004, Pages 162-168

[12] T.-R. Yang, and J.-J. Shieh, "Dynamic Fetch Engine Design for Simultaneous Multithreaded Processors," *In Proceedings of the 9th Asia-Pacific Computer Systems Architecture Conference*, Sep. 2004, Pages 489-502

[13] C.-H. Lin, and J.-J. Shieh, "A Study of Branch Prediction and Fetch Policy on Simultaneous Multithreading Architecture," *In Proceedings of the 9th World Multi-Conference on Systemics, Cybernetics and Informatics*, Orlando Florida, USA, July 2005

[14] P.-Y. Chang, M. Evers, and Y. Patt, "Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference," *1996 conference on Parallel Architectures and Compilation Techniques*, Oct. 1996, Pages 48-57

[15] P.M.W. Knijnenburg, A. Ramirez, F. Latorre, J. Larriba, and M. Valero, "Branch classification to control instruction fetch in simultaneous multithreaded architectures," *In 2002 International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, Jan. 2002, Pages 67-76

[16] T. Austin, E. Larson, D. Ernst, “SimpleScalar: an infrastructure for computer system modeling,” *IEEE Computer Journal*, Feb. 2002, Pages 59-67

[17] S. Hily, A. Seznec, “Branch Prediction and Simultaneous Multithreading,” *In Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, Oct. 1996, Pages 169-173