ABSTRACT

Title of thesis: STRATEGIES FOR ENHANCING THROUGHPUT AND FAIRNESS IN SMT PROCESSORS

Chungsoo Lim, Master of Science, 2004

Thesis directed by: Professor Manoj Franklin
Department of Electrical and Computer Engineering

Simultaneous Multithreading (SMT) is a technique to execute multiple threads in parallel in a single processor pipeline. An SMT processor has shared instruction queues and functional units and these resources are utilized efficiently without being wasted. Because the instruction queues and functional units are shared by multiple threads, it is very important to decide which threads to fetch instructions from every cycle.

This paper investigates 2-level fetch policies and other techniques with a view to improve both throughput and fairness. To measure the potential of the 2-level fetch policies, simulations are conducted on 4 different benchmark combinations with two SMT configurations, and simulation results are compared with those of ICOUNT and LC-BPCOUNT, two existing fetch policies. Our detailed experimental evaluation confirms that the 2-level fetch policies outperform both ICOUNT and LC-BPCOUNT in terms of throughput, as well as fairness.

As a way to improve fairness, we also investigate the idea of partially partitioning the instruction queues among the threads. In particular, we vary the partition size to see how both throughput and fairness are impacted. From this experiment, we found that more fairness can be obtained at the cost of throughput. We expect the techniques presented in this paper to play a major role in future SMT designs.

STRATEGIES FOR ENHANCING THROUGHPUT AND FAIRNESS IN SMT PROCESSORS

by

Chungsoo Lim

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
Of the requirements for the degree of
Master of Science
2004

Advisory Committee:
Professor Manoj Franklin, Chair
Professor Virgil Gligor
Professor Donald Yeung

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

A simultaneous Multithreading (SMT) processor is capable of executing multiple threads in parallel in a single pipeline [1] [2] [9]. The performance contribution of each thread in such a processor is dependent on the amount of resources allocated to it. As we increase the amount of resources allocated for a thread, its performance improves uniformly to a point, beyond which the improvement tends to be marginal. The fetch unit of the processor is in charge of controlling such resource allocation by slowing down or speeding up the fetch rate of specific threads.

Tellsen, et al [8] investigated several fetch policies for SMT processors. Among these, a policy called ICOUNT was found to provide the best performance. This policy gives the highest priority to the thread having the fewest instructions in the decode, rename, and instruction queue stages of the pipeline. The underlying assumption is that threads with fewer instructions in the pipeline are likely to make more efficient use of processor resources. However, this fetch policy doesn't take into account the probability of each thread to be in a wrong speculative path.

Luo, et al. [4] addressed this problem with a fetch policy (LC-BPCOUNT) that uses the number of outstanding low-confidence branch predictions to prioritize threads.

Threads with the fewest outstanding low-confidence branch predictions are given the highest priority (with pipeline instruction count used as a tie-breaker, when necessary). This fetch policy is likely to provide high throughput, as it assigns the processor resources to threads that are very likely to be in their correct paths. However, the LC-BPCOUNT policy does not do a good job on fairness. If a particular thread has many outstanding low-confidence branch predictions, that thread is likely to be consistently passed over in favor of threads that have fewer such predictions.

Both fairness and throughput are important when using an SMT processor. Whereas higher throughput ensures higher utilization of processor resources, fairness ensures that all threads are given equal opportunity and that no threads are forced to starve. Because different threads have different rates at which their instructions execute (and hence different native throughputs), prioritizing strictly on native throughput will result in giving priority to threads with high instruction execution rates all the time, causing the rest to suffer. In a multi-user environment, it is important to provide quality of service to all users. If we focus only on fairness, however, it can result in an inefficient use of processor resources, because a thread may often receive resources when it is least capable of utilizing it. Clearly, naïve approaches to increase throughput work against fairness, and vice versa.

This paper examines 2-level fetch policies for SMT processors, with special emphasis on enhancing both throughput and fairness. Our 2-level fetch policies prioritize based on both ICOUNT and LC-BPCOUNT. A 2-level prioritization policy, as the name implies, performs prioritization in 2 levels. One policy—either ICOUNT or LC-BPCOUNT—is used at the first level to partition the threads into several priority groups, and the other

one is used at the second level to further prioritize the threads inside each group. If we make use of good features from both the policies, better results can be obtained.

We also investigate a more intuitive way to improve fairness—partially partitioning the instruction queues. In this scheme, the instruction queues are divided into two regions, one of which is designed to be shared by all running threads, and the other is for private usage. This *private* region is partitioned according to the number of active threads, and each thread is given one partition for its own private use. Such a partial partitioning is expected to provide more fairness with some loss in throughput.

The rest of this paper is organized as follows. Section 2 presents background information on the SMT processor and SMT fetch policies. Section 3 discusses 2-level prioritizing for balancing throughput and fairness. Section 4 presents experimental results demonstrating the performance of our 2-level prioritization schemes. Section 5 presents partially partitioned instruction queues and their simulation results. Section 6 summarizes the paper.

Chapter 2

Background and Motivation

2.1 Simultaneous Multithreading Processor

The SMT processing engine, like most other processing engines, comprises of two major parts: the fetch engine and the execute engine. The fetch engine's objective is to fill the instruction queues (IQs) with instructions; it contains the I-cache, the branch predictor, the fetch unit, the decode unit, and the register rename unit, as shown in Figure 2.1. The execution engine's objective is to drain the instruction queues; it contains the instruction issue logic, the functional units, the memory hierarchy, the result forward mechanism, and the reorder buffer. The key features of the processor can be summarized as follows:

- The major resources are shared by all active threads.
- Every clock cycle, instructions from all active threads compete for each of the shared resources.
- Instructions of a thread are fetched and committed strictly in program order.
- Each of the resources in the processor has very limited buffering capabilities.
- Once an instruction enters the processor pipeline, it is not removed from the pipeline unless it is executed or found to be from an incorrect path. Therefore, it

is very important to select the best instructions every cycle at the fetch part of the pipeline where instructions enter the pipeline.

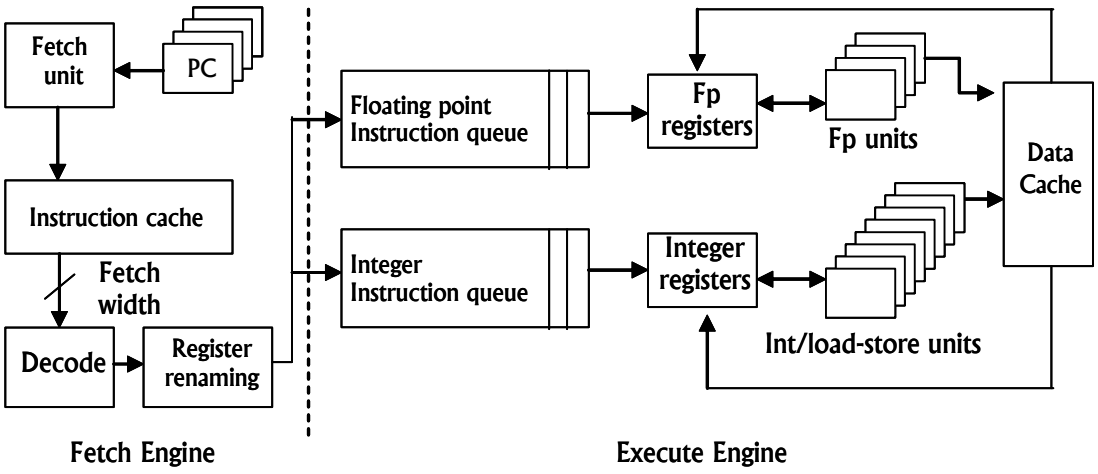


Figure 2.1: Block diagram of an SMT processor

2.2 Fetch Policies

Among the many fetch policies proposed so far, ICOUNT [8] and LC-BPCOUNT [4] have proven to be effective. We shall briefly describe them below.

2.2.1 ICOUNT (Instruction COUNT)

In this policy, the highest priority is given to those threads that have the fewest instructions in the pipeline (the decode stage, the rename stage, and the instruction queues). The motivation for this policy is two-fold: (i) give the highest priority to threads whose instructions are moving through the pipeline efficiently, and (ii) provide an even mix of instructions from the active threads. This naturally prevents any one thread from

monopolizing the instruction queues. However, it does not take into account whether or not a particular thread is in the correct path of execution.

2.2.2 LC-BPCOUNT (Low-Confidence Branch Prediction COUNT)

Because of incorrect branch prediction, a non-trivial portion of the instructions in an SMT pipeline can be from wrong paths. Wrong-path instructions not only have zero contribution to the throughput, but also waste valuable resources, preventing correct-path instructions of other threads from being fetched. The overall performance of SMT processors can be increased by reducing the number of incorrectly speculated instructions, in order to save resources for non-speculative or correctly speculated instructions. LC-BPCOUNT [4] attempts to do this. In this scheme, a confidence estimator [3] [5] is used to determine which predictions are likely to be correct and which predictions are likely to be incorrect. This estimator uses a table of miss distance counters (MDCs) to keep record of branch prediction correctness. Each MDC is a saturating resetting counter. A correctly predicted branch increments the corresponding MDC, whereas an incorrectly predicted branch resets its MDC to zero. Thus a high MDC value indicates a higher degree of confidence, and a low value indicates a lower degree of confidence. A branch is considered to have high confidence only when its MDC has reached a particular threshold value referred to as the *MDC-threshold*.

With this confidence information, the highest priority is given to the thread that has the fewest outstanding low-confidence (LC) branch instructions. Priority among threads having the same number of outstanding LC branches is determined by the threads' ICOUNT values.

Chapter 3

2-level Fetch Policies for Enhancing Throughput and Fairness

The policies discussed so far achieve high throughput or fairness by taking feedback from the processor pipelines and modifying the thread fetch priorities accordingly. Feedback from the pipeline is gathered in terms of system variables such as outstanding low-confidence branch count and instruction count. To achieve high throughput as well as fairness, we propose combining the two system variables.

3.1 Basic Idea

The instantaneous value of a pipeline system variable (such as instruction count) can be used to assign fetch priorities to threads. In order to obtain high throughput as well as fairness, we may need to look at multiple system variables at the same time. When looking at multiple variables, there must be some means to integrate them. One way to do this is to assign weights for each system variable and to calculate an overall fetch priority for each thread. The drawback of this approach is that it requires the fetch unit to perform complex calculations every cycle. A more plausible approach is to perform prioritization in multiple steps or levels, with each level involving a different system variable. That is, first use one of the variables to partition the threads into different

priority groups (with members of each group having equal priority in terms of that variable), and then use another variable to further prioritize the threads inside each group, and so on.

Threads fall into the same priority group if their system variables under consideration (either instruction count or low-confidence branch prediction count) are in the same range; we call the range size as the *partitioning granularity*. These ranges cover all possible values that a system variable can have, and there is no overlap between adjacent ranges. Notice that if the *partitioning granularity* is set to the maximum value of the system variable under consideration, there is just one big range and all threads fall into the same priority group, regardless of what values they have for that system variable. If the *partitioning granularity* is set to 1, on the other extreme, each thread may belong to its own priority group. In this case, the second-level prioritization is useless.

The working of the 2-level policy is illustrated in Figure 3.1. The *partitioning granularity* of the first level is set to 8 in this example. There are five threads, whose initial priority order is the same as the numerical order. The threads are prioritized at the first level based on the instruction counts shown in the figure, creating three priority groups. Because the *partitioning granularity* is 8, the first range is from 0 to 7, the second range covers 8-15, and so on. Therefore, threads {0, 1}, and {2, 4} are grouped together, respectively. Based on the outstanding low-confidence branch prediction counts, second-level prioritization is performed inside each priority group. Let's look at the first priority group. As thread 0's outstanding low-confidence branch prediction count is lower than that of thread 1, no change is made to the order inside the first priority group.

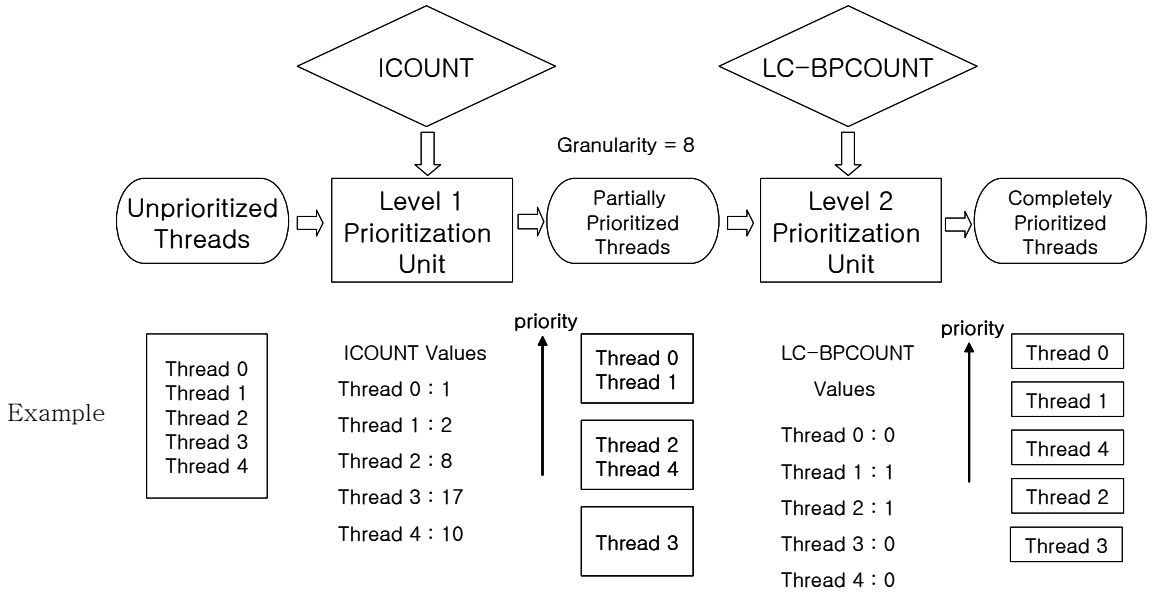


Figure 3.1: 2-level prioritization when the *partitioning granularity* of the first level is 8

On the other hand, when we look at the second group, thread 4 has a higher priority than thread 2 after the second-level prioritization, because thread 4 has fewer outstanding low-confidence branch predictions. Note that thread 2 would have had higher priority than thread 4, if ICOUNT alone had been used. This illustrates how the second-level policy makes a difference. Unlike thread 4, thread 3 doesn't have a chance to have a higher priority because its instruction count is too big to be in the same group as thread 2.

3.2 Hardware Requirements

Let's look at the extra hardware required for the 2-level prioritization schemes. Figure 3.2 represents an implementation of a 2-level prioritization scheme, which uses ICOUNT at the first level and LC-BPCOUNT at the second level. The *partitioning granularity* is set to 8 as before. The first prioritization unit partitions the threads into several priority groups, according to their instruction count values, and the second-level

prioritization units perform prioritization inside each group, according to the outstanding low-confidence branch prediction counts.

If we set the *partitioning granularity* to a power of 2, the first-level grouping can be done very easily in hardware: the priority group number will be given by the bits remaining after removing the least significant $\log_2$(*partitioning granularity*) bits from the ICOUNT value. In the example presented in Figure 3.2, we need to check only the 3 most significant bits out of the 6 bits that represent the ICOUNT value of a thread (because the least significant $\log_2 8=3$ bits are removed). If these bits are '000', the corresponding thread ID is assigned to priority group 0, which covers instruction counts 0 through 7. Likewise, '001' is for priority group 1.

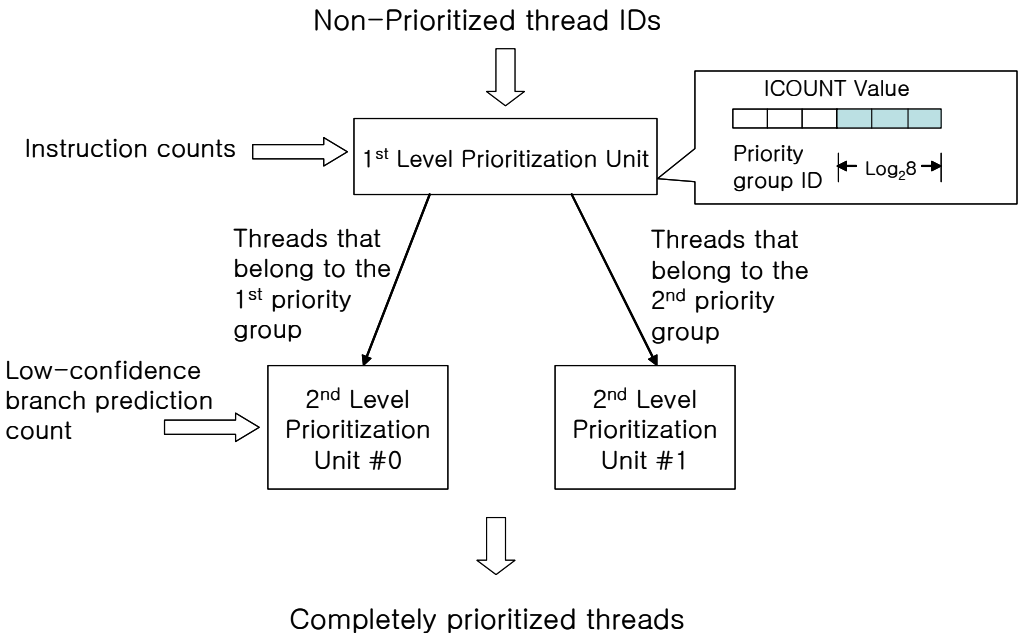


Figure 3.2: A more detailed view of 2-level prioritization when the *partitioning granularity* is 8

If the maximum instruction count allowed is 48, there should be 6 prioritization units at the second level. However, our experiments show that only two prioritization units are sufficient when the *partitioning granularity* is set to 8. This is because the instruction count of a thread usually falls in the range 0 to 15. When there are threads with instruction counts more than 15, the number of such threads is likely to be small, and these threads are unlikely to be selected as the thread to fetch new instructions from. As we can see, the extra hardware requirement is quite moderate.

Chapter 4

Experimental Evaluation

We have presented the concepts of 2-level prioritization and its hardware requirements. This section presents a detailed experimental evaluation of these 2-level prioritization schemes.

4.1 Evaluation Setup

The experiments in this section are conducted using a simulator derived from the public domain SMT simulator developed by Tullsen, et al [8]. This simulator executes Alpha object code, and models an SMT processor. Some of the simulator parameters are set as follows. The instruction pipeline has 9 stages (based on the Alpha 21264 pipeline), but includes extra cycles for accessing a large register file. Functional unit latencies are also from the Alpha 21264 processor. The memory hierarchy has 64 KB 2-way set-associative instruction and data caches, a 1024 KB 2-way set associative on-chip L2 cache, and a 4MB off-chip cache. All cache line sizes are 64 bytes. All of the on-chip caches are 8-way banked. Cache miss penalties are 6 cycles to L2 cache, another 12 cycles to the L3 cache, and another 62 cycles to the main memory. The fetch unit fetches a maximum of *f* instructions in a cycle from up to 2 threads, where *f* is the fetch size. The

second priority thread gets an opportunity only if *f* instructions could not be fetched from the first priority thread due to events such as instruction cache misses. We simulate two different fetch sizes: 8 and 16. For the confidence estimator of the LC-BPCOUNT policy, the MDC-threshold is set to 8.

Workload

The workload used in the simulations consists of 11 programs, 5 of which are from the SPEC95 integer benchmark suite (Compress95, Cc, Go, Li, Ijpeg) and the rest from the SPEC2000 integer benchmark suite (Gzip, Bzip, Gcc, Eon, Parser, Twolf). These programs have different individual IPC values, branch misprediction rates, and instruction cache hit rates. Each data point is collected by first fast forwarding the benchmarks and then simulating the SMT processor for a total of 500 million instructions (excluding wrong-path instructions and flushed instructions). The extent of fast forwarding is as done in [10], and is presented in Table 4.1.

SPEC95 Benchmark	Number of instructions Fast Forwarded	SPEC2000 Benchmark	Number of instructions Fast Forwarded
Compress95	0.2 billion	Gzip	0.1 billion
Cc	0.2 billion	Bzip	0.1 billion
Go	0.2 billion	Gcc	0.5 billion
Li	0.2 billion	Eon	1 billion
Ijpeg	0.2 billion	Parser	0.5 billion
		Twolf	1 billion

Table 4.1: The number of instructions emulated before beginning measured simulation

The following four benchmark combinations are used in the evaluation:

1) **Combination B1** : {Compress, Go, Cc, Ijpeg, Li} - this combination consists of only SPEC 95 integer programs.

2) **Combination B2** : {Go, Cc, Gcc, Parser, Twolf} - this combination is a mix of programs from both SPEC 95 and SPEC 2000. All programs in this combination have low branch hit rate and low native IPC.

3) **Combination B3** : {Compress, Ijpeg, Bzip, Eon, Li} - this combination is a mix of programs from both SPEC 95 and SPEC 2000. All programs in this combination have high branch hit rate and high native IPC.

4) **Combination B4** : {Gzip, Gcc, Eon, Parser, Twolf} - this combination consists of only SPEC 2000 integer programs.

SMT Configurations Simulated

We simulated the following two SMT configurations:

Configuration C1: 32-slot IQ, 6 integer functional units (4 of which can perform load/store), 3 floating point units, and a fetch bandwidth of 8 instructions per cycle.

Configuration C2: 64-slot IQ, 12 integer functional units (8 of which can perform load/store), 6 floating point units, and a fetch bandwidth of 16 instructions per cycle.

Fetch Policies Simulated

We simulate the following 4 fetch policies. The basic ICOUNT policy ignores the probability for threads to be in wrong paths, and the basic LC-BPCOUNT policy does not limit the number of instructions in the pipeline from a thread.

1. **ICOUNT** : basic ICOUNT policy. If there is a tie between two threads, the priority information for the previous cycle is used.
2. **LC-BPCOUNT** : basic LC-BPCOUNT policy. If there is a tie, the ICOUNT value is used to break the tie.
3. **(I-LCBP)COUNT** : granularity-based 2-level prioritization policy, with ICOUNT at the first level and LC-BPCOUNT at the second level. The granularity governs how the two policies are mixed.
4. **(LCBP-I)COUNT** : granularity-based 2-level prioritization policy, with LC-BPCOUNT at the first level and ICOUNT at the second level.

4.2 Metrics

We use the following metrics to quantify the throughput and fairness associated with each fetch policy.

1. *Throughput*: Throughput is measured in terms of IPC (instructions per cycle). We measure the IPC of each thread, as well as the overall IPC of the SMT processor. Also, we use the *weighted speedup* metric introduced by Tullsen, et al. [10]. With this metric,

results are calculated by dividing a thread's IPC by its IPC in the baseline configuration with the same set of threads. The definition of *weighted speedup* is:

$$Weighted\ Speedup = \frac{1}{N} \sum_{Threads} \frac{IPC_{new}}{IPC_{baseline}}$$

where N is the total number of running threads. This metric distinguishes true speedup from false speedups, which are obtained by simply favoring high-IPC threads.

2. *Thread fetch priority*: This metric measures the average fetch priority given to each thread, with 0 being the highest priority.

3. *Individual thread speedups*, *their standard deviation, and harmonic mean*: individual thread speedup is the speedup experienced by each thread, with respect to the throughput it obtained when executed in single thread mode. If all threads experience similar speedups—meaning that the standard deviation of individual thread speedups is small—then we may conclude that the fetch policy is fair (although the threads may not have had the same fetch priority). The standard deviation doesn't account for throughput at all. However, the harmonic mean of individual thread speedups can encapsulate both throughput and fairness. The harmonic mean tends to be lower if one or more threads have lower speedup, thereby capturing some effect of the lack of fairness.

4.3 Throughputs in Single Thread Mode

In the first set of experiments, we simulate each benchmark in single thread mode for 100 million instructions, after fast forwarding the respective number of instructions. (As each combination mentioned in Section 4.1 is made up of 5 benchmarks and each simulation runs for 500 million instructions in SMT mode, roughly 100 million

instructions are executed from each thread when run in the multithreading mode.) Single-thread throughput results are useful when analyzing the results presented in the later sections.

Table 4.2 presents for each benchmark program, the fraction of instructions that are branches, the branch prediction accuracy, and the IPC (instruction per cycle). The fraction of instructions that are branches, and the branch prediction accuracy, have some bearing on the number of outstanding low-confidence branch predictions.

Program	Fraction of instructions that are branches (%)	Branch prediction accuracy (%)	IPC
Compress	6.68	89.1	3.02
Go	9.52	74.8	2.00
Cc	11.30	84.5	1.91
Ijpeg	5.52	88.6	3.60
Li	12.27	94.6	3.07
Gzip	8.54	89.4	2.91
Gcc	10.32	84.9	1.82
Parser	10.22	92.0	2.12
Twolf	9.67	83.6	1.61
Bzip	9.73	96.2	2.47
Eon	7.57	93.1	2.87

Table 4.2: Benchmark characteristics

Note that Ijpeg has the fewest fraction of branch instructions, a high branch prediction accuracy, and the highest native IPC. Ijpeg will therefore play a significant role in LC-BPCOUNT's performance.

4.4 Results for 2-level Prioritization Fetch Policies

The next set of experiments evaluates the performance of 2-level prioritization techniques. The four benchmark combinations discussed in Section 4.1 are used in this simulation, with SMT configuration C1 (also described in Section 4.1). The simulations were done by varying the *partitioning granularity* from 1 to 32. As the granularity is increased, the influence of first-level prioritization becomes weaker. Results for both the IPC and the standard deviation of average fetch priority are presented.

4.4.1 Results for Benchmark Combination B1

Figure 4.1 shows the results for benchmark combination B1. Part (a) shows throughput and (b) shows fairness. X-axis denotes the *partitioning granularity* in both cases. There are 4 curves in each figure, corresponding to the 4 fetch policies. Notice that the curves for ICOUNT and LC-BPCOUNT are horizontal lines, as they are not dependent on the *partitioning granularity*.

(I-LCBP)COUNT :

Let's look at the curves for (I-LCBP)COUNT. When the *partitioning granularity* is small, both the IPC and the standard deviation of average fetch priority are similar to those of ICOUNT. As the *partitioning granularity* increases, the IPC increases to a peak and then starts to decrease to the IPC of LC-BPCOUNT, and the standard deviation decreases below that of ICOUNT and rises to that of LC-BPCOUNT. The peak IPC is higher than those of ICOUNT and LC-BPCOUNT.

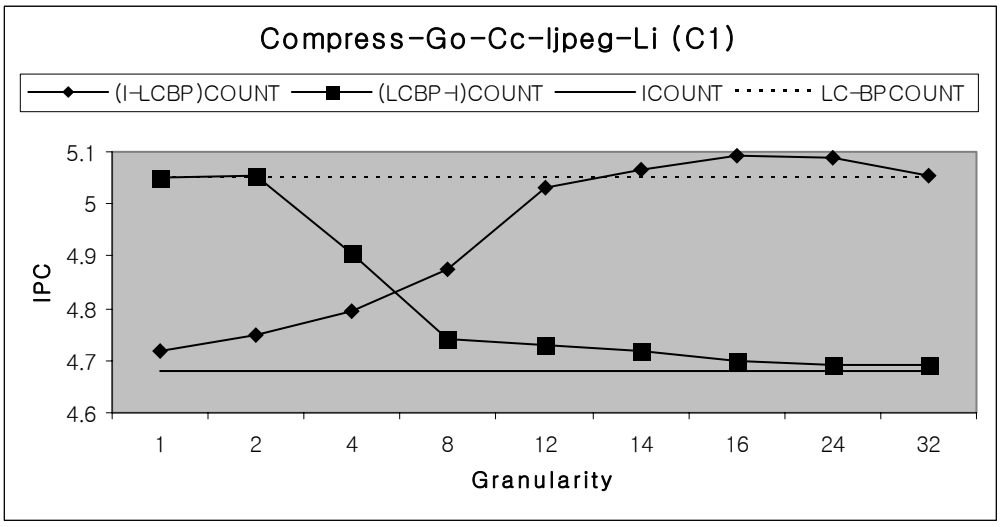


Figure 4.1 (a): Throughput for benchmark combination B1 with machine configuration C1

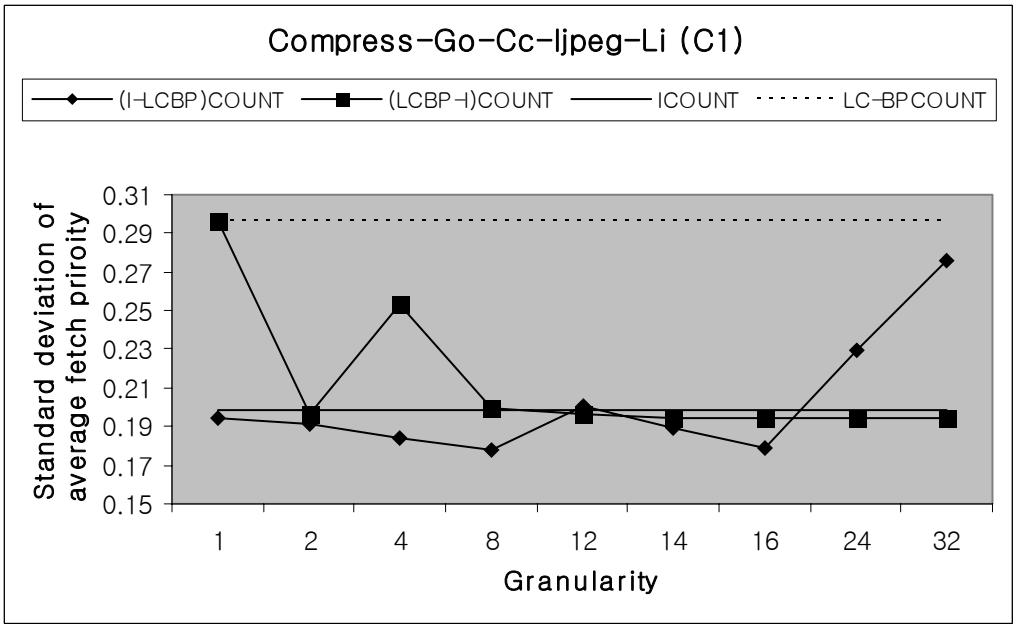


Figure 4.1 (b): Fairness for benchmark combination B1 with machine configuration C1

Conceptually, when LC-BPCOUNT is used at the second level, some threads that have high branch prediction accuracy are likely to be given higher priority than when ICOUNT is solely used, resulting in higher IPC. On the other hand, threads with low branch prediction accuracy are likely to be given lower priority than when only ICOUNT is used, resulting in lower IPC. This tendency becomes strong as the *partitioning granularity* becomes very large. The gain obtained by threads that have high branch prediction accuracy should be more than the loss experienced by threads that have low branch prediction accuracy in order for (I-LCBP)COUNT to have a higher overall IPC than that of ICOUNT. To a certain point, the gain is higher than the loss. After this peak IPC point, the gain is lower than the loss. Consequently, the throughput keeps decreasing, until it converges to the IPC of LC-BPCOUNT.

Let's look at what is happening at *partitioning granularity* 16, where (I-LCBP)COUNT is even better than LC-BPCOUNT in terms of throughput. All the benchmarks except Ijpeg (which has the best branch characteristics (low branch count and high branch prediction accuracy) and the highest native throughput), are given higher priority than when LC-BPCOUNT is used alone. Consequently, every benchmark other than Ijpeg produces a higher throughput than when LC-BPCOUNT is used alone. The decrease in throughput from Ijpeg is less than the increase from the other benchmarks, and so the overall throughput of (I-LCBP)COUNT is more than that of LC-BPCOUNT at *partitioning granularity* 16. If we look at this from the perspective of wrong-path instructions and instruction queue capacity conflicts (instruction queue conflict is measured as the number of cycles when new instructions cannot be enqueued because the instruction queues are full), the instruction queue capacity conflicts of (I-LCBP)COUNT

(11.1%) is less than that of LC-BPCOUNT (14.5%), while the fraction of wrong-path instruction of (I-LCBP)COUNT (12.6%) is more than that of LC-BPCOUNT (10.6%). Fewer instruction queue capacity conflicts seem to indicate that instructions are clearing the pipeline at a faster rate.

Even when a thread is given a lower priority than when ICOUNT is used, the thread is able to produce higher throughput than that with ICOUNT. This is possible when the throughput loss due to lowered priority is less than the gain from reduction in wrong-path instructions. For larger *partitioning granularities,* the loss of throughput due to lowered priority overruns the gain.

For standard deviation of average fetch priority (see Figure 4.1(b)), the same behavior is witnessed. When the *partitioning granularity* is small, the standard deviation is closer to that of ICOUNT and starts to follow that of LC-BPCOUNT. The two 2-level fetch policies are fairer than both ICOUNT and LC-BPCOUNT at some granularities. For this benchmark combination, the priorities of some threads that are higher than the average are decreasing and the priorities of other threads that are lower than the average are increasing, as the *partitioning granularity* increases. Therefore, the standard deviation of the priorities is getting smaller to a certain point. After that point, the standard deviation starts to increase as the priorities keep changing.

(LCBP-I)COUNT :

Next, let's look at the curves for (LCBP-I)COUNT. The throughput as well as the fairness of (LCBP-I)COUNT are almost the same as those of LC-BPCOUNT, when the *partitioning granularity* is 1. On the other hand, when the *partitioning granularity* is

large, the throughput and fairness of (LCBP-I)COUNT are close to those of ICOUNT. As the *partitioning granularity* is increased, the effect of LC-BPCOUNT reduces, and the wrong-path instructions effectively increase. This causes a drop in throughput.

At *partitioning granularity* 2, the fairness of (LCBP-I)COUNT improves dramatically to that of ICOUNT, while maintaining the high throughput achievable by LC-BPCOUNT. Because the number of outstanding low-confidence branch predictions is normally small, the fairness of (LCBP-I)COUNT tends to be sensitive to small changes in *partitioning granularity*. For higher *partitioning granularities*, (LCBP-I)COUNT does not appear to be useful.

At *partitioning granularity* 4, the standard deviation of average fetch priority of (LCBP-I)COUNT sticks out like a sore thumb. This is because the average priority of each thread may not always change in one direction as the *partitioning granularity* changes. This irregular behavior entails the abnormality. This erratic behavior is caused because we have multiple active threads competing for more hardware resources. Let's say a thread is given less hardware resources as the *partitioning granularity* increases, and 4 threads are competing for more hardware resources. Because the rate of increase of thread priority differs from one to another, it is possible for a thread to obtain lower priority than what was obtained with smaller granularities. The rate depends on the system variables under consideration (ICOUNT and LC-BPCOUNT in our 2-level prioritization schemes).

4.4.2 Results for Benchmark Combination B2

Figure 4.2 presents simulation results for benchmark combination B2. For this combination, ICOUNT has higher throughput than LC-BPCOUNT, but is less fair than LC-BPCOUNT! This is because this benchmark combination has no program that has good branch characteristics and high native throughput such as Ijpeg. Because there is no such dominating thread that is likely to be favored by LC-BPCOUNT, LC-BPCOUNT can be fairer than ICOUNT. However, it should be noted that the difference between ICOUNT and LC-BPCOUNT in terms of fairness is less than the difference seen for benchmark combination B1. The 2-level policies are not fairer than LC-BPCOUNT for most of the granularities.

(I-LCBP)COUNT :

Let's look at the curves for (I-LCBP)COUNT. At *partitioning granularity* 12, (I-LCBP)COUNT is at its peak in terms of throughput. Every benchmark obtains higher throughput than when ICOUNT is used. For those threads with worse branch characteristics, there are two reasons why even they produce higher throughput than when ICOUNT is solely used. The first reason is that because there is no dominating benchmark, the degree of priority change due to LC-BPCOUNT at the second level is relatively small, compared to that for benchmark combination B1. Therefore, the throughput decrease from the threads with worse branch characteristics is small. The second reason is the reduction in wrong-path instructions caused by employing LC-BPCOUNT at the second level.

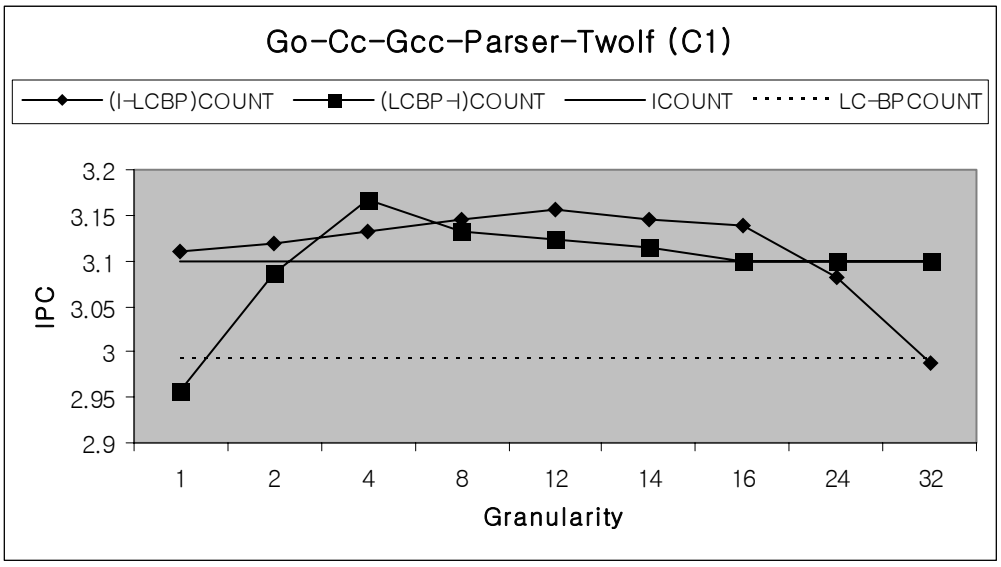


Figure 4.2 (a): Throughput for benchmark combination B2 with machine configuration C1

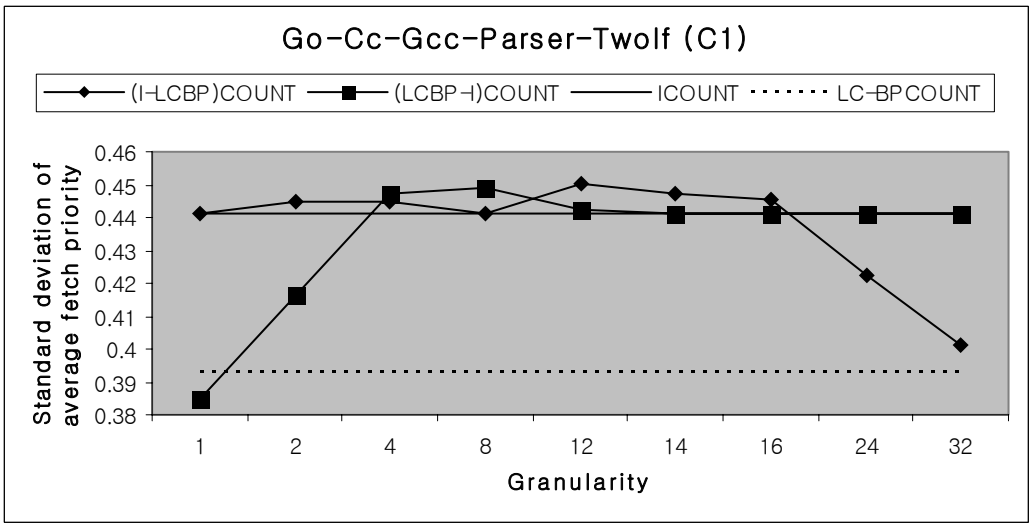


Figure 4.2 (b): Fairness for benchmark combination B2 with machine configuration C1

As the *partitioning granularity* increases past the peak throughput point, the throughput decrease for the benchmarks with bad branch characteristics is more than the throughput increase for the benchmarks with good branch characteristics. This is because the native throughputs of the threads with relatively good branch characteristics (Parser, Twolf) in this benchmark combination are not big enough to offset the throughput decrease. Consequently, the throughput goes down to that of LC-BPCOUNT.

The standard deviation of average fetch priority fluctuates around that of ICOUNT until the *partitioning granularity* reaches 16, and then converges to that of LC-BPCOUNT. The priority of each thread is changing either for fairness or against fairness with different speed (this also varies over granularity) as the *partitioning granularity* increases. Consequently the standard deviation fluctuates until it starts to converge to that of LC-BPCOUNT. During this fluctuation, the standard deviation may be higher or lower than that of ICOUNT. It depends on the characteristics of the benchmarks being simulated.

(LCBP-I)COUNT :

For (LCBP-I)COUNT, the throughput reaches its maximum value when the *partitioning granularity* is 4. At this granularity, all benchmarks produce higher throughput than that of ICOUNT. A couple of threads receive higher priority than with ICOUNT, so that they obtain a higher throughput. For those threads that receive lower priority than with ICOUNT, the same reasons as for (I-LCBP)COUNT apply here. The difference in priority is small (throughput decrease due to lowered priority is small) and the gain from reduced wrong-path instructions result in a higher throughput.

Note that, at *partitioning granularity* 2, the throughput of (LCBP-I)COUNT improves from that of LC-BPCOUNT, maintaining its fairness above that of ICOUNT (smaller standard deviation of average fetch priority than that of ICOUNT). This indicates that (LCBP-I)COUNT is capable of balancing throughput and fairness.

4.4.3 Results for Benchmark Combination B3

Figure 4.3 shows the results for benchmark combination B3. LC-BPCOUNT has higher throughput and less fairness than ICOUNT due to Ijpeg and Bzip, which are the two dominating benchmarks in this combination. Because there are two dominating benchmarks, the curves in Figure 4.3 are similar to those for benchmark combination B1. Note that at *partitioning granularity* 2, (LCBP-I)COUNT produces higher throughput than LC-BPCOUNT. What is happening is that the increase in throughput from the threads with bad branch characteristics due to increased priorities are bigger than the loss in throughput from the threads with good branch characteristics due to decreased priorities and increased number of wrong-path instructions.

The fairness of both 2-level prioritization policies are better than that of ICOUNT at some granularities. At *partitioning granularity* 8, the throughput of (I-LCBP)COUNT is more than that of LC-BPCOUNT, while fairness also surpasses that of ICOUNT. At *partitioning granularity* 2, (LCBP-I)COUNT is fairer than LC-BPCOUNT, while its throughput exceeds that of LC-BPCOUNT.

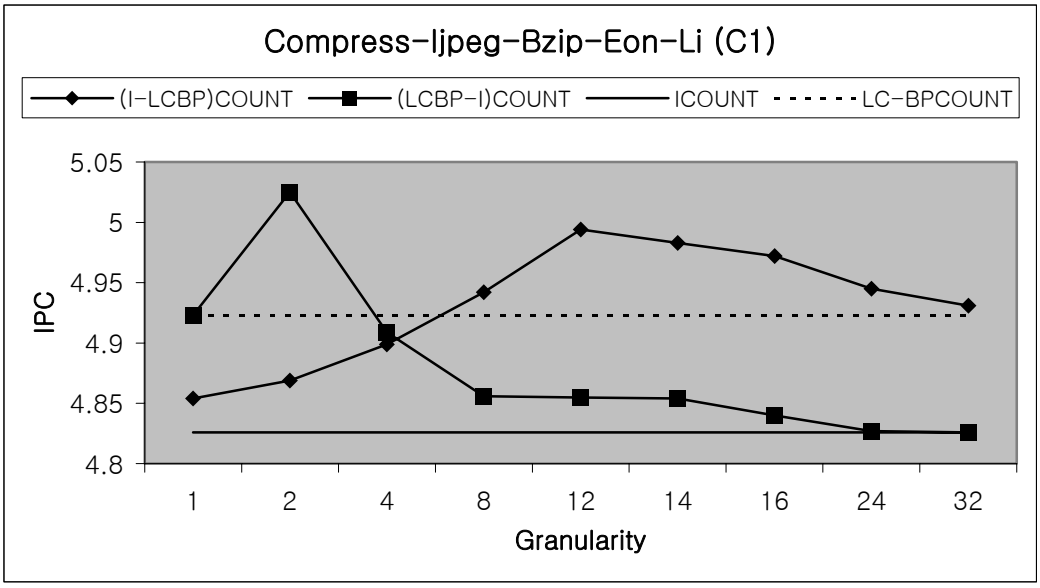


Figure 4.3 (a): Throughput for benchmark combination B3 with machine configuration C1

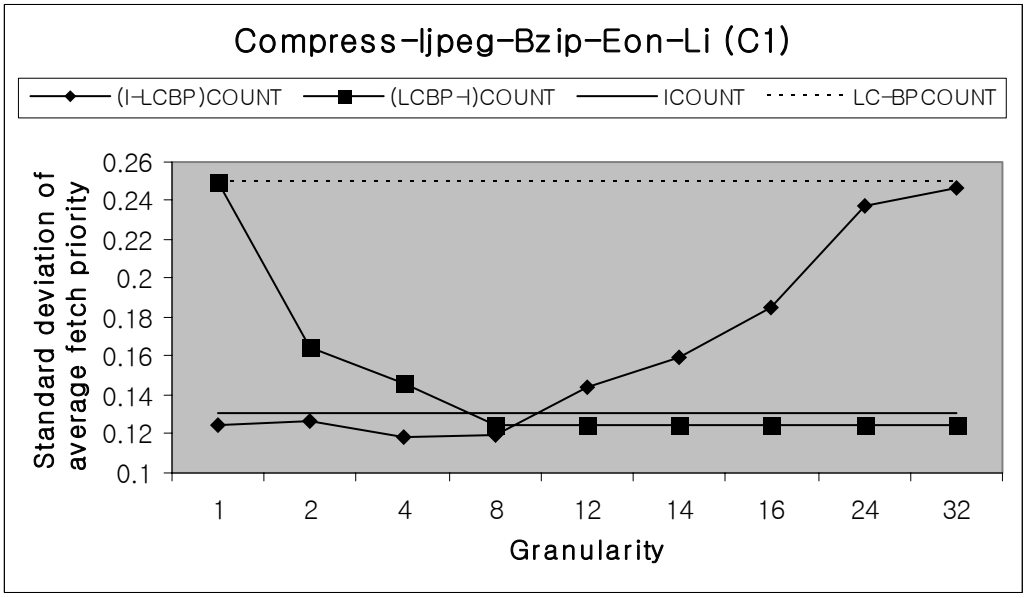


Figure 4.3 (b): Fairness for benchmark combination B3 with machine configuration C1

4.4.4 Results for Benchmark Combination B4

Figure 4.4 presents the results for benchmark combination B4. This combination is similar in characteristics to combination B2. ICOUNT is better than LC-BPCOUNT in terms of throughput, and LC-BPCOUNT is better than ICOUNT when it comes to fairness. Again, both of the 2-level prioritization policies are not as fair as LC-BPCOUNT.

The capability of the 2-level prioritization schemes in balancing throughput and fairness can also be proven here by looking at the results for *partitioning granularities* 24 and 2 for (I-LCBP)COUNT and (LCBP-I)COUNT, respectively.

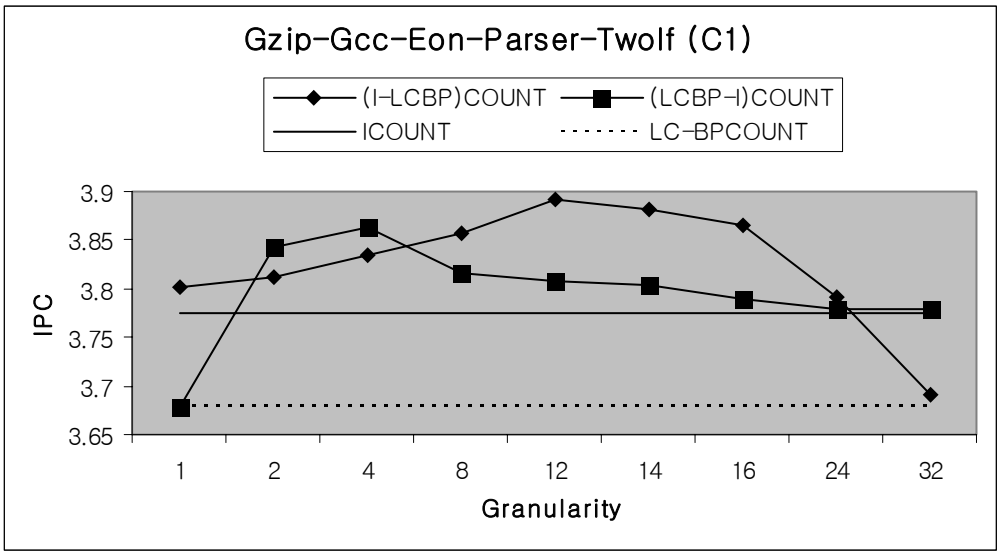


Figure 4.4 (a): Throughput for benchmark combination B4 with machine configuration C1

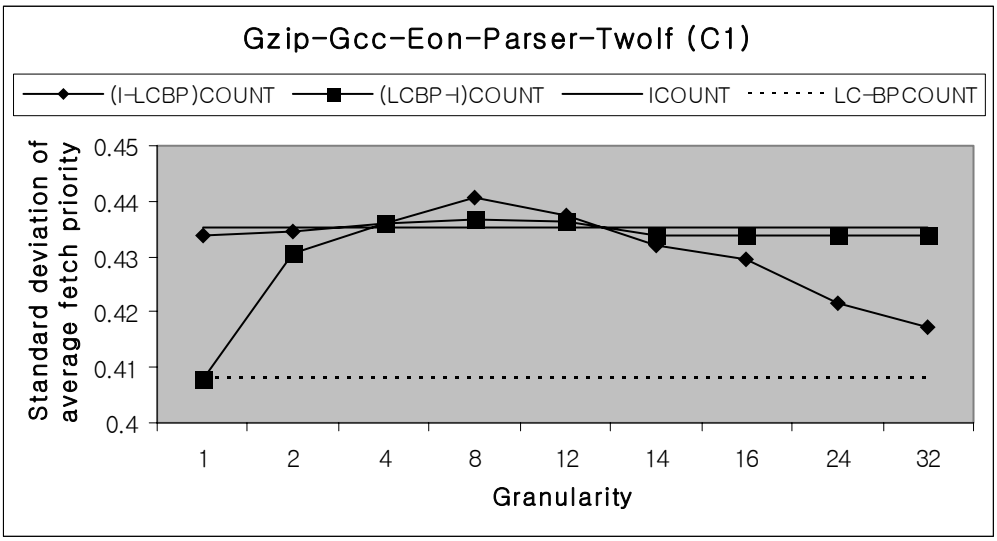


Figure 4.4 (b): Fairness for benchmark combination B4 with machine configuration C1

4.4.5 Summary

We saw seen that the 2-level fetch policies are able to produce higher throughput than both ICOUNT and LC-BPCOUNT. Generally, ICOUNT is able to provide more parallelism to the instruction queues than LC-BPCOUNT, and LC-BPCOUNT is capable of reducing the number of wrong-path instructions. With the 2-level fetch policies, the highest throughput is achieved when the optimal tradeoff between the amount of parallelism and the number of wrong-path instructions is obtained.

For fairness, the common behavior is that the standard deviation starts very close to that of the fetch policy at the first level and starts to converge to that of the fetch policy at the second level. The *partitioning granularity* at which the convergence starts depends on the characteristics of the benchmarks. Usually, convergence starts at *partitioning granularity* 2 for (LCBP-I)COUNT and at *partitioning granularity* 16 for (I-LCBP)COUNT. Until the standard deviation starts to move toward that of the fetch

policy at the second level, fluctuation of the standard deviation is observed. During this fluctuation, the standard deviation is either higher or lower than that of the fetch policy at the first level.

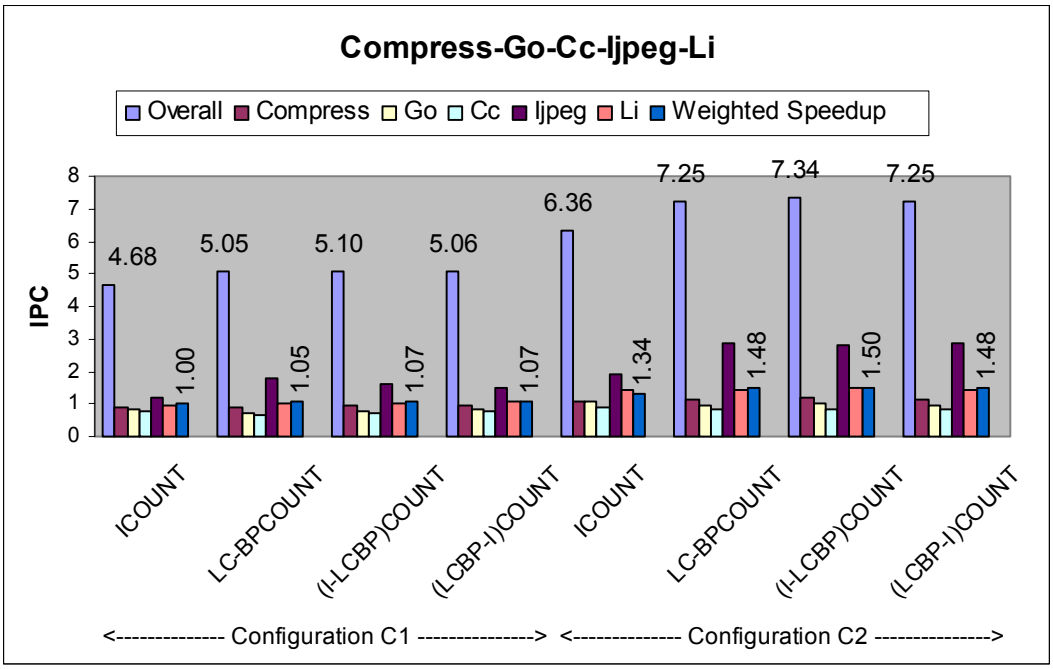
We can also observe that if the *partitioning granularity* is set to 14, or 16 for SMT configuration C1, (I-LCBP)COUNT is capable of generating higher IPC than both ICOUNT and LC-BPCOUNT. In addition to IPC, low standard deviation of average fetch priority can also be achieved at the granularities. If the *partitioning granularity* is set to 2, (LCBP-I)COUNT is able to produce higher throughput than both ICOUNT and LC-BPCOUNT.

4.5 Individual throughputs and weighted speedup

Next, let us look at the individual throughputs obtained in SMT mode using the 4 fetch policies. By comparing several fetch policies for four different benchmark combinations, it is possible to pinpoint the best fetch policy in terms of throughput. For these experiments, we use both the SMT configurations, C1 and C2, mentioned in Section 4.1. The metrics used in this section are IPC and *weighted speedup,* as described in Section 4.1. The *partitioning granularities* used for the 2-level prioritization policies are those that produced the highest IPCs.

Figure 4.5 presents the results for combination B1. The Y-axis indicates the IPC and the X-axis lays out the different fetch policies. Each fetch policy has 7 histogram bars: the first one indicates the overall throughput, and the next five correspond to the IPC contributions of the five individual threads, and the last bar indicates the *weighted speedup.* The weighted speedup is calculated using SMT configuration C1 with ICOUNT

as the baseline policy.



(I-LCBP)COUNT is the best for both configurations in terms of both overall IPC and *weighted speedup*. This means that (I-LCBP)COUNT is able to mix ICOUNT and LC-BPCOUNT policies judiciously such that optimal tradeoff between the throughput gain by the threads with good branch characteristics and the throughput loss by the threads with bad branch characteristics is found. Note that even though the overall IPCs of LC-BPCOUNT and (LCBP-I)COUNT are very close (for SMT configuration C1), the *weighted speedup* of (LCBP-I)COUNT is higher than that of LC-BPCOUNT. This means that (LCBP-I)COUNT is fairer than LC-BPCOUNT in terms of resource allocation.

Figure 4.6 shows the throughput results for benchmark combination B2. Again, (I-LCBP)COUNT and (LCBP-I)COUNT produce the highest overall throughput and *weighted speedup*. For this combination, the throughput of LC-BPCOUNT is less than

that of ICOUNT, as seen earlier. The fact that LC-BPCOUNT has lower IPC than ICOUNT means that outstanding low-confidence branch prediction count is overused, so that the benefit from reduced number of wrong-path instructions is overcome by a lack of parallelism in the pipeline, which could've been more, had ICOUNT been used. In other words, the throughput of (I-LCBP)COUNT is the output when outstanding low-confidence branch prediction count is more adequately utilized.

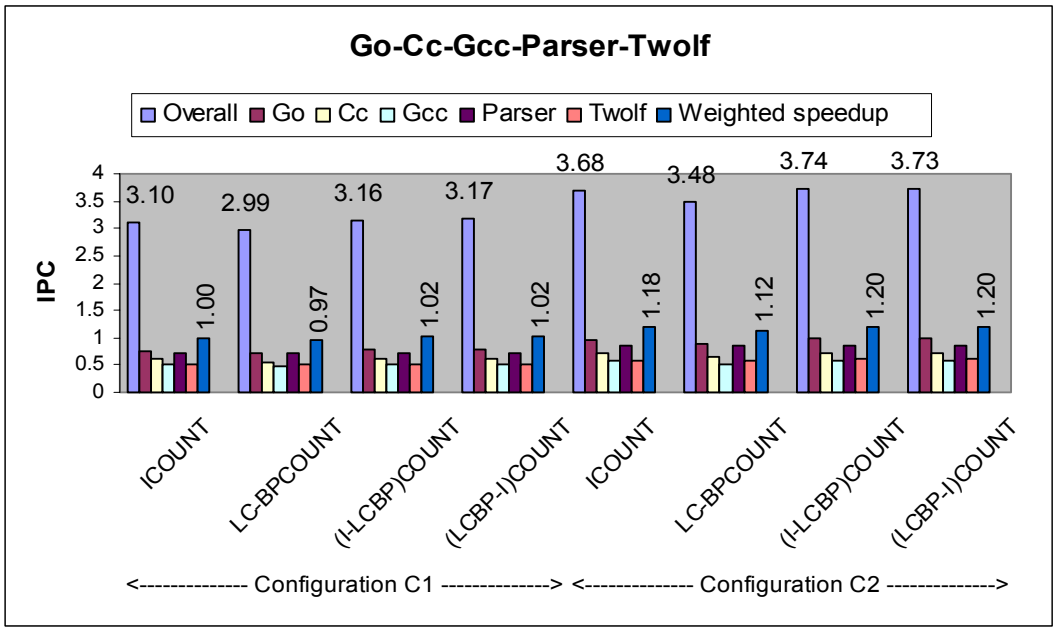
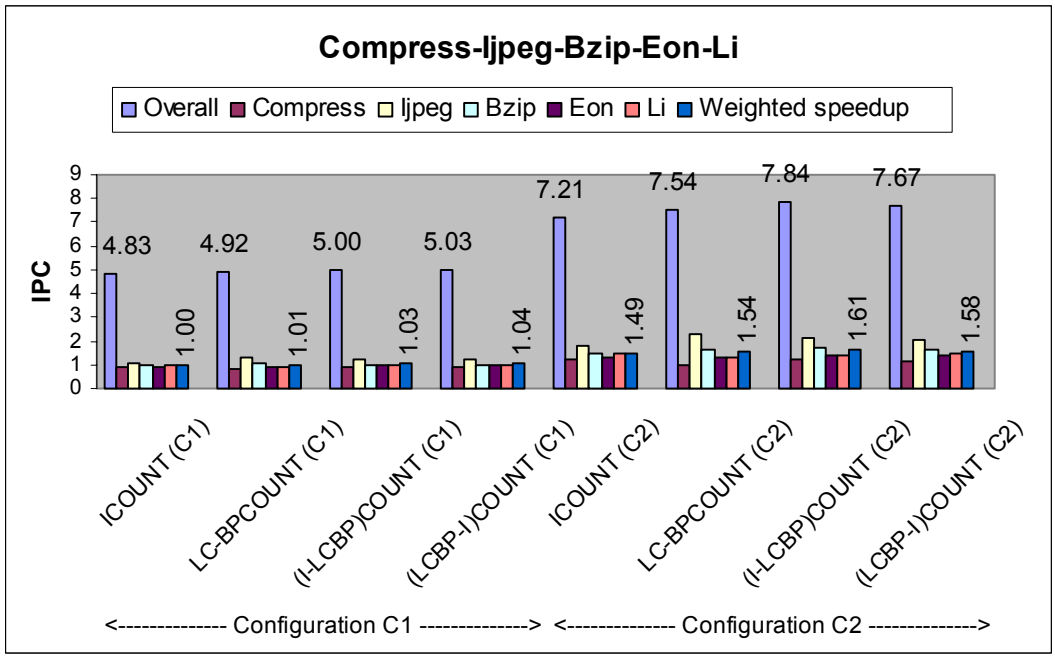


Figure 4.6: Throughput for benchmark combination B2

Figure 4.7 and 4.8 presents the results for benchmark combinationB3 and 4 respectively. 2-level prioritization schemes outperform ICOUNT and LC-BPCOUNT. Explanation on these benchmark combinations are omitted because benchmark combinationB3 and 4 are similar in behavior to benchmark combinationB1 and 2 respectively.


Compress-ljpeg-Bzip-Eon-Li
Overall
Compress
ljpeg
Bzip
Eon
Li
Weighted speedup
IPC
4.83
1.00
4.92
1.01
5.00
1.03
5.03
1.04
7.21
1.49
7.54
1.54
7.84
1.61
7.67
1.58
ICOUNT (C1)
LC-BPCOUNT (C1)
(I-LCBP)COUNT (C1)
(LCBP-I)COUNT (C1)
ICOUNT (C2)
LC-BPCOUNT (C2)
(I-LCBP)COUNT (C2)
(LCBP-I)COUNT (C2)
<---------- Configuration C1 ----------> <---------- Configuration C2 ---------->
Gzip-Gcc-Eon-Parser-Twolf
overall
Gzip
Gcc
Eon
Parser
Twolf
Weighted speedup
IPC
3.78
1.00
3.68
0.96
3.89
1.02
3.89
1.02
4.81
1.25
4.62
1.19
4.95
1.27
4.93
1.27
ICOUNT (C1)
LC-BPCOUNT (C1)
(I-LCBP)COUNT (C1)
(LCBP-I)COUNT (C1)
ICOUNT (C2)
LC-BPCOUNT (C2)
(I-LCBP)COUNT (C2)
(LCBP-I)COUNT (C2)
<---------- Configuration C1 ----------> <---------- Configuration C2 ---------->
Figure 4.7: Throughput for benchmark combination B3
Figure 4.8: Throughput for benchmark combination B4

From the throughput results presented, we can see that the 2-level fetch policies are able to produce the highest throughput if appropriate *partitioning granularity* is selected. As mentioned in Section 4.4, selecting the right *partitioning granularity* is not difficult. *Partitioning granularity* 14 and 2 work well for all benchmark combinations with (I-LCBP)COUNT and (LCBP-I)COUNT respectively, even though they may not be the optimal *partitioning granularity* at which the maximum throughput is produced, for every benchmark combination.

4.6 Individual Thread Fetch Priorities

Next, we shall look at the detailed results for the average fetch priority obtained by the threads. Note that a lower value for the average fetch priority indicates higher priority. Standard deviation of average fetch priority is also reported, as before. The lower the standard deviation obtained by a policy is, the fairer the policy is. The *partitioning granularities* used for the 2-level prioritization policies are those that achieved the lowest standard deviations for average fetch priority.

Figure 4.9 shows the results for combination B1. The first five bars for each fetch priority indicate the average fetch priorities measured for the 5 threads, and the sixth bar indicates the standard deviation of the average fetch priorities. For SMT configuration C1, (I-LCBP)COUNT is the best policy. For C2, the standard deviations for the 2-level prioritization policies are just a little bit higher than that of ICOUNT but far less than that of LC-BPCOUNT. Unlike LC-BPCOUNT, (I-LCBP)COUNT doesn't favor one particular benchmark, and introducing the low-confidence branch prediction count improves the fairness in this case.

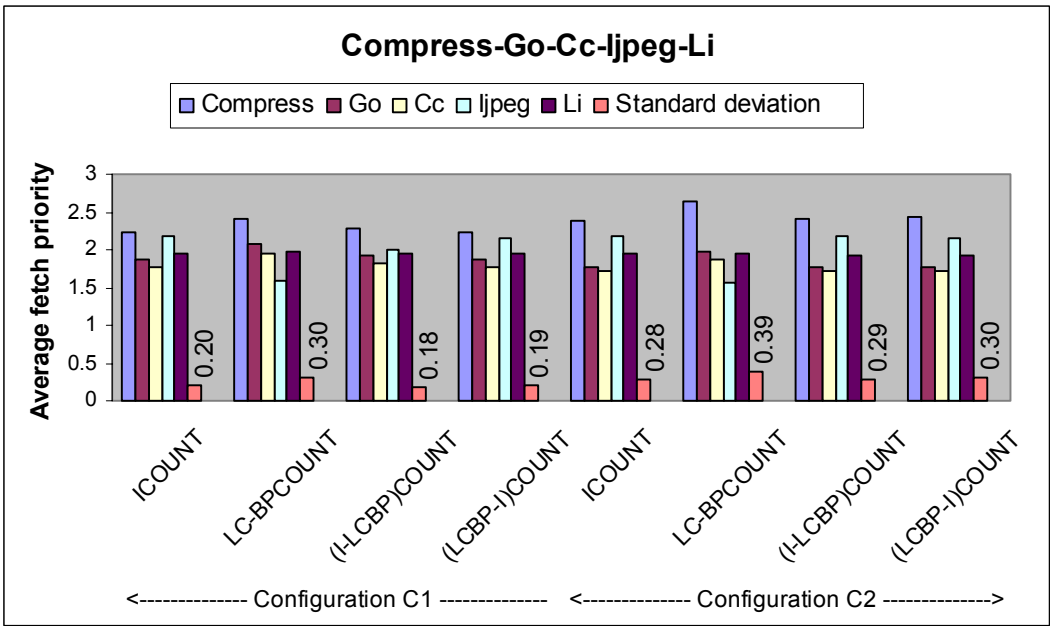
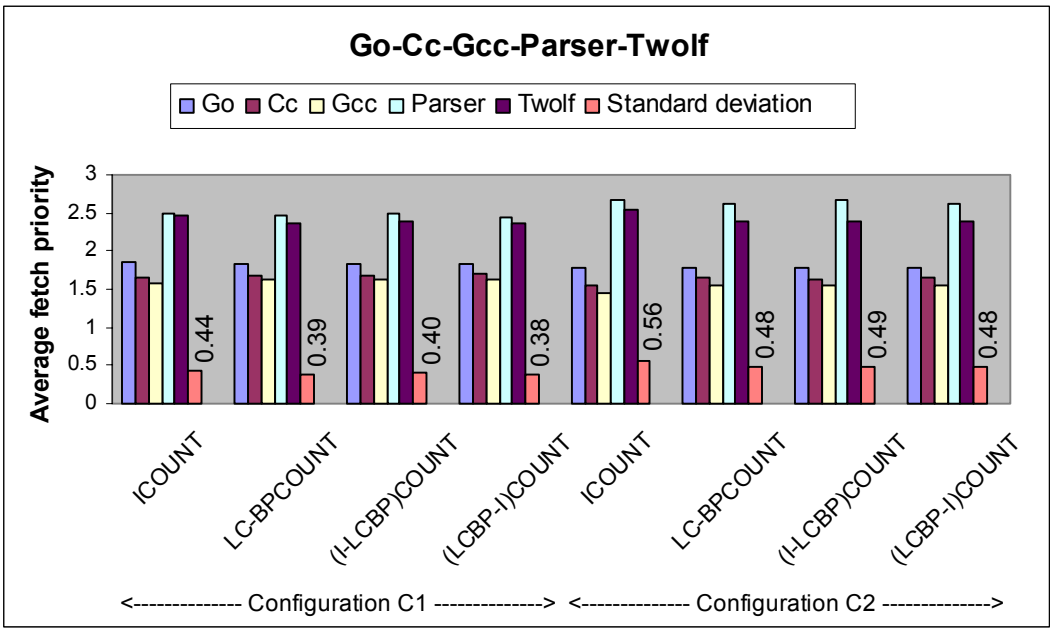


Figure 4.10 presents the results for benchmark combination B2. (LCBP-I)COUNT is the best one here. Note that LC-BPCOUNT beats ICOUNT. Generally, LC-BPCOUNT is considered to be very poor in fairness, but whether LC-BPCOUNT is fair or not depends on the branch characteristics (branch count and branch prediction accuracy) of each thread. However, it should be noted that when LC-BPCOUNT is fair to all threads, the overall throughput is likely to be lower than that of ICOUNT.



The results for benchmark combinationB3 are presented in Figure 4.11. 2-level fetch policies are fairer than any other policies. Figure 4.12 has simulation results for benchmark combinationB4. (LCBP-I)COUNT is the best policy that provides the most fairness. (LCBP-I)COUNT improves the fairness of LC-BPCOUNT by incorporating instruction count. This incorporation happens to improve the fairness of LC-BPCOUNT.

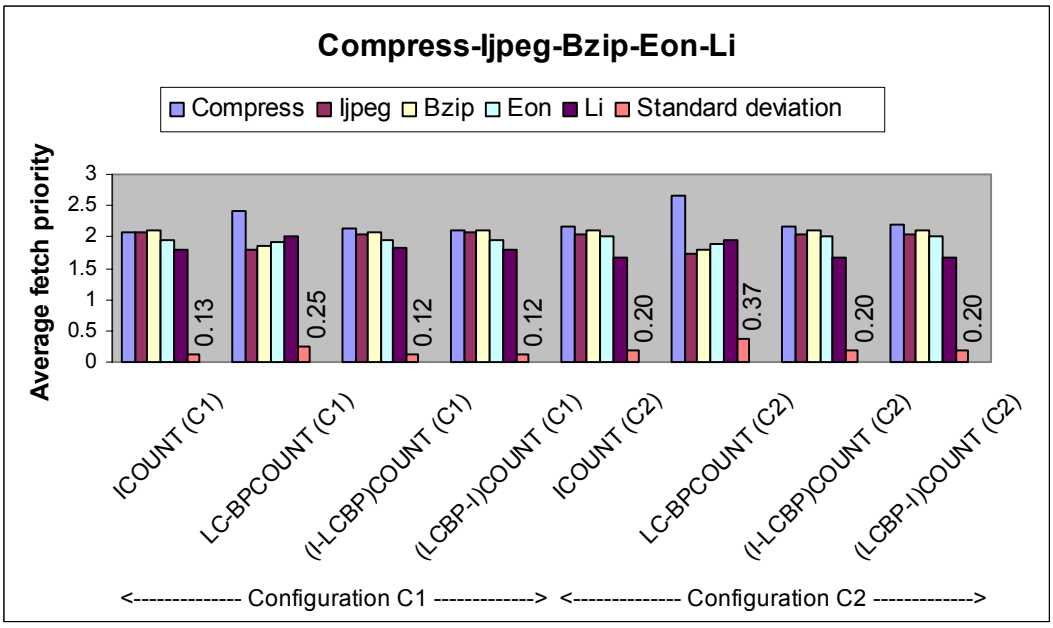


Figure 4.11: Average fetch priority for benchmark combination B3

Figure 4.12: Average fetch priority for benchmark combination B4

From this set of simulations, it can be seen that by setting the *partitioning granularity* properly, it is possible for the 2-level fetch policies to become as fair as whichever is better (between ICOUNT and LC-BPCOUNT). This is because the 2-level fetch policies can be very close to either ICOUNT or LC-BPCOUNT by setting the *partitioning granularity* very small or very large. It may be possible for the 2-level fetch policies to be fairer than ICOUNT and LC-BPCOUNT, if the second-level policy alleviates the unfairness caused by the first-level policy.

4.7 Individual Thread Speedups

The metrics used so far in the earlier sections primarily focus on either throughput or fairness. In this section, a metric that addresses both throughput and fairness is used. This is the harmonic mean of the relative throughputs obtained for the individual threads compared to their throughputs in single-thread mode. Considering the relative throughputs helps to factor out the inherent discrepancies between the different threads. The *partitioning granularities* used for the 2-level prioritization policies are those that achieved the highest values for the harmonic means of the relative throughput. Note that the *partitioning granularities* for highest harmonic means coincide with those for highest throughput.

Figures 4.13 through 4.16 present the relative IPC results for benchmark combinations B1, B2, B3 and B4, respectively. In these figures, the first five bars indicate the relative IPC values of the five threads; the sixth bar indicates the harmonic mean of the relative IPCs of the five threads, and the last bar represents the standard deviation of the five relative IPC values. Notice that whereas the standard deviation of the average fetch

priority (used earlier) represents how equally threads are given priority, the standard deviation of the relative throughput indicates how well a fetch policy conforms to the amount of parallelism each thread has. In other words, low standard deviation of relative throughput means that it is more likely for a thread with more parallelism to get higher priority and for a thread with less parallelism to get lower priority.

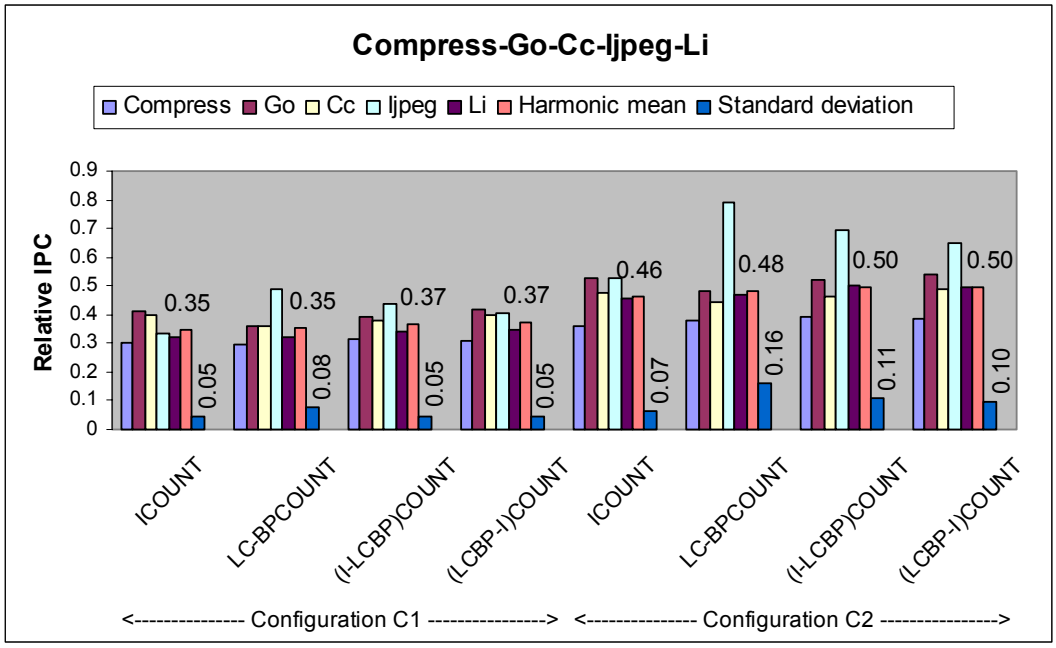


Figure 4.13: Relative IPC for benchmark combination B1

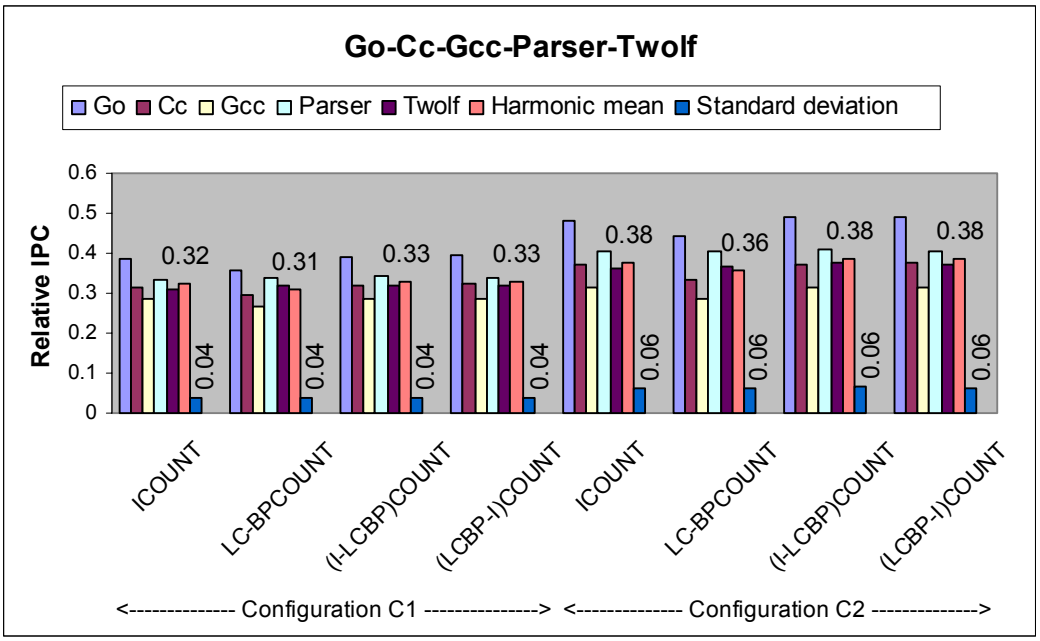


Figure 4.14: Relative IPC for benchmark combination B2

Figure 4.15: Relative IPC for benchmark combination B3

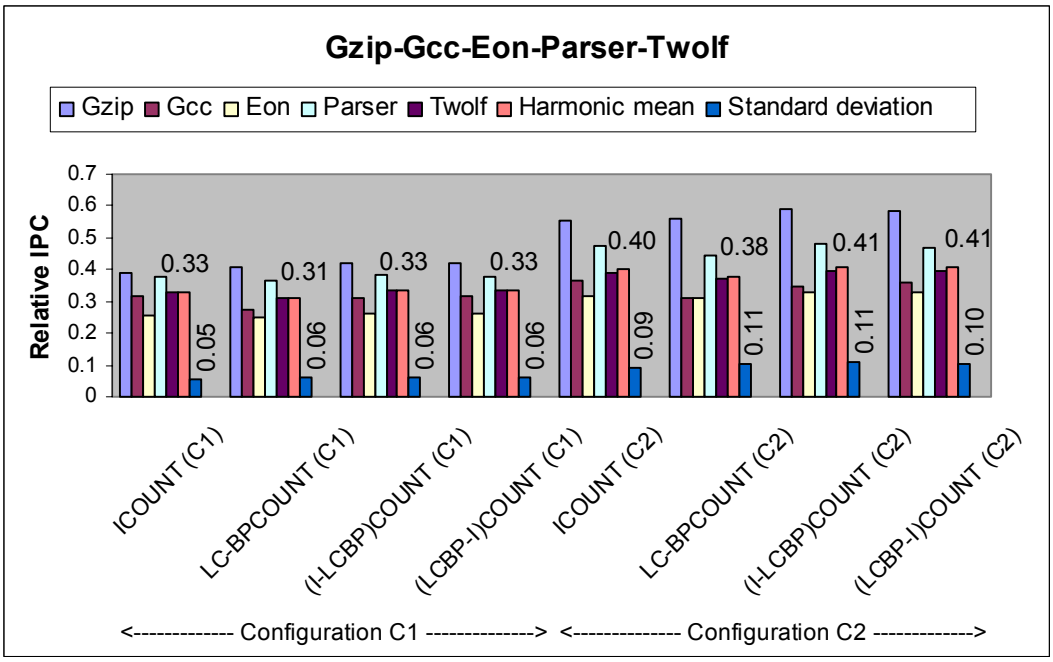


Figure 4.16 Relative IPC for benchmark combination B4

Considering the harmonic mean, the 2-level fetch policies are the best. This means they can balance throughput and fairness better than both ICOUNT and LC-BPCOUNT.

If we look at the standard deviation of relative IPC, ICOUNT outperforms the other policies. This reveals that ICOUNT works the way it is supposed to. Recall that one of the motivations for ICOUNT described in Section 2.2.1 is to give the highest priority to threads whose instructions are moving through the pipeline efficiently. The 2-level fetch policies are just a little bit behind ICOUNT. From this set of results, it is observed that the 2-level fetch policies have the ability to balance throughput and fairness, giving instantaneous priorities to threads according to their native parallelism.

Chapter 5

Partially Partitioned Instruction Queue

The 2-level fetch policies discussed so far achieve high throughput, while maintaining a decent amount of fairness. If we are more interested in fairness, as a way to achieve fairness besides fetch policies, partially partitioning the instruction queues among the active threads may be a good solution.

5.1 Basic Idea

When an instruction queue is shared by multiple threads, fairness among threads can be impaired, as discussed earlier. The degree of unfairness depends on the fetch prioritization policy. On one extreme, we can provide maximum fairness by using the round robin fetch policy. However, the round robin fetch policy drastically reduces the throughput.

In this section, we investigate partially partitioned instruction queues as an effort to find a trade-off between throughput and fairness. By partially partitioning the instruction queue, the dominant thread is suppressed and wasted empty slots can be partially eliminated.

A partially partitioned instruction queue consists of two regions, a *private* queue

region and a *shared* queue region. The *private* region is divided into smaller queues, each of which accommodates a single assigned thread. Every active thread has its own *private* queue and can occupy slots in the *shared* queue if it needs more space. In this way, a limited degree of unfairness is allowed for maintaining good throughput, while fairness is improved to some extent. For example, let's suppose there are 32 slots in an instruction queue, and 4 threads are running. One possible configuration (geared for fairness) is to give each thread 7 *private* slots, allowing the remaining 4 slots to be shared by all threads. On the other hand, a configuration geared for throughput might assign 2 private slots to each thread and reserve 24 slots for the *shared* region. The extent of partitioning can even be dynamically changed, according to the nature of the threads.

5.2 Experimental Results

Figure 5.1 represents the simulation results we obtained for the partially partitioned instruction queue. SMT configuration C1 is used in this simulation, and the number of slots assigned to each thread for private use is varied from 0 to 6. When 6 private slots are assigned to each thread, 30 slots are allocated for private use, leaving only two slots for shared use. This is one extreme case where the instruction queue is almost partitioned according to the number of threads. When the number of private slots assigned to each thread is 2, only 10 slots are reserved for private use, leaving 22 slots for shared use. Note that there are many cycles when no instruction is fetched at all because fetch/decode and decode/register-rename stages are not empty. Fetch is blocked until these pipeline registers are empty. These pipeline registers cannot be cleared when there is no space in instruction queues for the particular threads that the instructions in the

pipeline registers belong to.

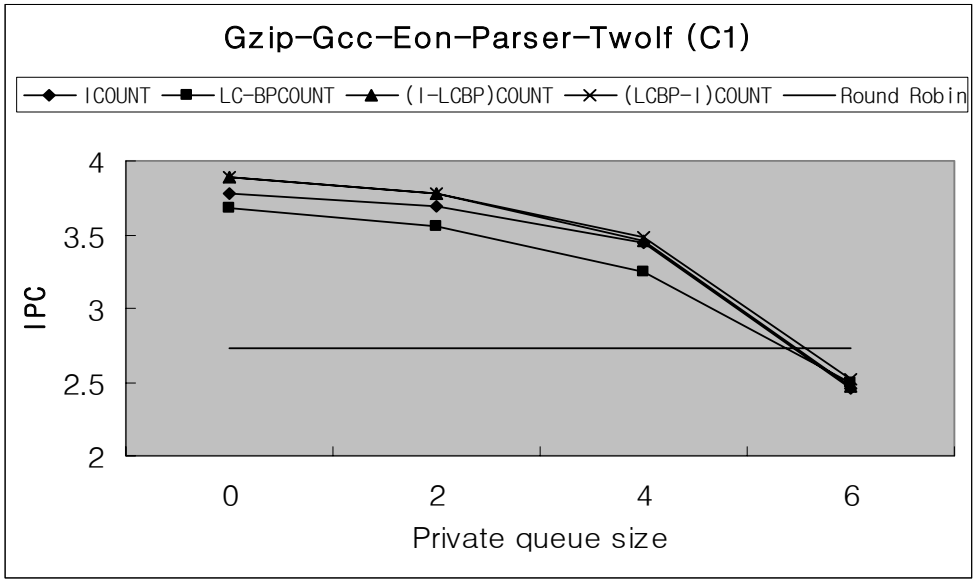


Figure 5.1 (a): Impact of partition size on throughput

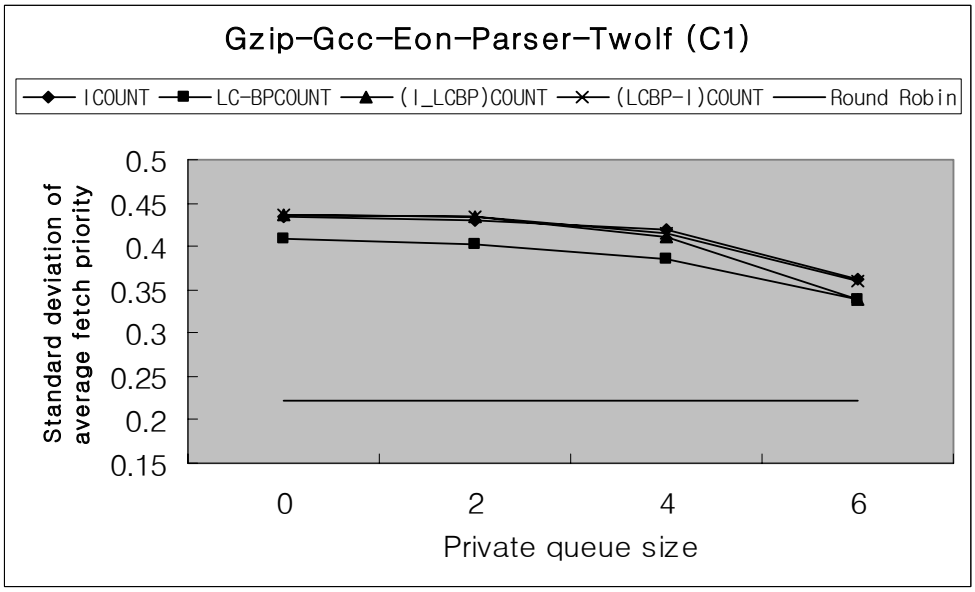


Figure 5.2 (b): Impact of partition size on standard deviation of average fetch priority

The round robin policy, which is known to be the best policy for fairness, is also included in the figure to see how well the partially partitioned instruction queues work. Note that the results for the round robin policy are without the partially partitioned instruction queues.

The results show that partially partitioned instruction queues can reduce the standard deviation of average fetch priority, but at the expense of throughput. As expected, if we increase the number of slots assigned for private use, fairness increases and throughput decreases. For *private* queue size 2 and 4, there is no significant change in both fairness and throughput. In order to enhance the fairness by a non-trivial amount, the *private* queue size should be at least 4 with which some amount of throughput is lost. Note that at *private* queue size 6, round robin outperforms the other fetch policies in terms of both throughput and fairness. From this simulation, it has been shown that in order to achieve ample fairness gain, employing round robin policy is the best way.

Chapter 6

Summary and Conclusion

Simultaneous Multithreading (SMT) permits multiple threads to execute in parallel within a single processor. Usually, an SMT processor uses shared instruction queues to schedule instructions from the different threads. Hence an SMT processor's performance depends on how the instruction fetch unit fills these instruction queues. Each cycle, the fetch unit must carefully decide which threads to fetch instructions from.

This paper addressed the issue of enhancing both throughput and fairness by using 2-level fetch policies that utilize pipeline system variables, and by using partially partitioned instruction queues. Metrics that measure throughput, fairness, and both were used to evaluate the results.

In order to measure throughput, IPC and *weighted speedup* were used and for fairness, the standard deviation of average fetch priority was used. To encapsulate both throughput and fairness, harmonic mean of relative speedups was also used. In order to compare the 2-level fetch policies with existing policies such as ICOUNT and LC-BPCOUNT, 4 benchmark combinations were simulated with two different SMT configurations. Our focus was placed on both throughput and fairness.

Our simulation results reveal that (I-LCBP)COUNT (a 2-level prioritization scheme

with ICOUNT at the first level and LC-BPCOUNT at the second level) is the best policy in terms of throughput. (I-LCBP)COUNT outperforms ICOUNT by 4.6% for SMT configuration C1, and by 7.2% for SMT configuration C2. Furthermore, this fetch policy surpasses LC-BPCOUNT by 3.7% for SMT configuration C1, and by 5% for SMT configuration C2. This enhancement occurs when the *partitioning granularity* is set to the value that allows the proper degree of help from LC-BPCOUNT. The fairness at this *partitioning granularity* is not the best that (I-LCBP)COUNT can achieve, but at least it is better than the worst one.

If we consider fairness, the 2-level fetch policies perform well. For the 2 benchmark combinations for which ICOUNT is fairer than LC-BPCOUNT, the 2-level prioritization policies are the best. For the other 2 benchmark combinations, for which LC-BPCOUNT is fairer than ICOUNT, the 2-level policies are very close to LC-BPCOUNT in terms of fairness. It should be noted that for the benchmark combinations for which ICOUNT is fairer than LC-BPCOUNT, the best fairness is achieved at the *partitioning granularity* near the one at which the highest throughput is obtained, and that for the benchmark combinations for which LC-BPCOUNT is fairer than ICOUNT, the best fairness is achieved at very small or big *partitioning granularity* for (LCBP-I)COUNT or (I-LCBP)COUNT, respectively.

When we consider the harmonic mean of individual thread speedups, the 2-level policies stand at the top of the list. This shows that the 2-level fetch policies are second to none, as far as balancing throughput and fairness is concerned. By setting the *partitioning granularity* appropriately, it is possible to achieve decent throughput and fairness at the same time. As mentioned in Section 4.7, the throughput at this *partitioning*

granularity happens to be the highest. The fairness at this granularity may not be the best, but no other policy achieves this amount of throughput and fairness at the same time.

The 2-level prioritization schemes presented in this paper are versatile. They are capable of improving throughput, fairness, or balancing throughput and fairness. Because the extra hardware required for the schemes is very significant, the 2-level fetch policies have the potential to replace existing policies.

Our experimental evaluation also showed that partially partitioned instruction queues are able to achieve better fairness, but by sacrificing throughput. It has been confirmed that partially partitioning the instruction queues is generally a bad idea for achieving fairness.

References

[1] G. E. Daddis, Jr. and H. C. Torng, "The Concurrent Execution of Multiple Instruction Streams on Superscalar Processors," *Proc. International Conference on Parallel Processing (ICPP)*, pp. I:76-83, 1991.

[2] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous Multithreading: A Foundation for Next-generation Processors," *IEEE Micro*, pp. 12-18, September/October 1997.

[3] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun, "Confidence Estimation for Speculation Control," *Proc. 25th International Symposium on Computer Architecture*, 1998.

[4] K. Luo, M. Franklin, S. S. Mukherjee, and A. Seznec, "Boosting SMT Performance by Speculation Control," *Proc. 15th International Parallel & Distributed Processing Symposium (IPDPS)*, 2001.

[5] E. Jacobsen, E. Rotenberg, and J. E. Smith, "Assigning Confidence to Conditional Branch Predictions," *Proc. 29th International Symposium on Microarchitecture (MICRO-29)*, pp.142-152, December 1996.

[6] C. McFarling, "Combining Branch Predictors," *WRL Technical Note TN-36*, June 1993.

[7] A. Snavely and D. M. Tullsen, "Symbiotic Job Scheduling for a Simultaneous Multithreading Processor", *Proc. 9th International Conference on Architectural Support*

for Programming Languages and Operating Systems (ASPLOS-IX), 2000.

[8] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *Proc. 23rd International Symposium on Computer Architecture*, pp. 191-202, May 1996.

[9] W. Yamamoto and M. Nemirovsky, "Increasing Superscalar Performance Through Multistreaming," *Proc. IFIP WG10.3 Working Conference on Parallel Architectures and Complilation Techniques (PACT'95)*, pp. 49-58, 1995.

[10] K. Luo, J. Gummaraju, and M Franklin, "Balancing Thoughput and Fairness in SMT Processors," *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2001.

[11] D. M. Tullsen, and J. A. Brown, "Handling Long-latency Loads in a Simultaneous Multithreading Processor." *Proc 34th International Symposium on Microarchitecture (MICRO-34)*, December, 2001

[12] L. Kleinrock, Queuing Systems, Vol. 1. Wiley: New York, 1975.

[13] D. Ortega, I.Martel, E.Ayguade, M. Valero, and V. Venkat, "A Characterization of Parallel SPECint Programs in Simultaneous Multithreading Architectures," *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, 1999.

[14] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22nd International Symposium on Computer Architecture*, June, 1995.

[15] A. El-Moursy, and D. H. Albonesi, "Front-End Policies for Improved Issue

Efficiency in SMT Processors," *Proc. 9th International Symposium on High-Performance Computer Architecture (HPCA-9'03)*, 2002.

[16] R. Kessler. "The Alpha 21264 microprocessor," *IEEE Micro*, 19(2): 24-36, March/April 1999.

[17] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Qualifying the Complexity of Superscalar Processors," *Technical Report CS-TR*-96-1328, 1996.

[18] S. Sair, and M. Charney, "Memory Behavior of the SPEC2000 Benchmark Suite," *Technical Report, IBM Corporation*, Oct 2000.