# Boosting SMT Performance by Speculation Control

Kun Luo Manoj Franklin
ECE Department
University of Maryland
College Park, MD 20742, USA
{kunluo, manoj}@eng.umd.edu

Shubhendu S. Mukherjee
334 South St, SHR3-2/R28
Compaq Computer Corp.
Shrewsbury, MA 01545, USA
shubu.mukherjee@compaq.com

Andre Sezne
campus de Beaulieu
IRISA/INRIA
35042 Rennes Cedex, France
seznec@irisa.fr

## Abstract

*Simultaneous Multithreading (SMT) is a technique that permits multiple threads to execute in parallel within a single processor. Usually, an SMT processor uses shared instruction queues to collect instructions from the different threads. Hence, an SMT processor's performance depends on how the instruction fetch unit fills these instruction queues. On each cycle the fetch unit must judiciously decide which threads to fetch instructions from.*

*This paper proposes a new instruction fetch scheme that uses both* fetch prioritizing *and* fetch gating *for SMT processors. Fetch prioritizing sets up fetch priority for each thread based on the number of unresolved low-confidence branches from the thread, while fetch gating prevents fetching from a thread once it has a stipulated number of outstanding low-confidence branches. Based on the fetch priority of each thread, our fetch scheme finds threads that are most likely to be in their correct paths. This improves the overall throughput of an SMT processor by reducing the number of wrong-path instructions in the pipeline. Our experimental evaluation shows that, on the average, our fetch scheme provides 14.9% speedup over ICOUNT, which is the best fetch policy reported so far for SMT.*

## 1. Introduction

Simultaneous multithreading (SMT) is a recently proposed multithreaded processor design in which multiple thread contexts are active simultaneously [1] [2] [8] [10]. The active thread contexts typically share all resources in an SMT processor. The instructions per cycle (IPC) contribution of each thread is dependent on the amount of resources available to the thread. Unfortunately, the relationship between a thread's IPC and amount of allocated resources is rarely linear. As a thread receives more resources, its IPC increases somewhat uniformly up to a point, beyond which the increase tends to be marginal. Interestingly, the instruction fetch unit, which supplies instructions from different threads to the SMT processor, can control such resource allocation by slowing down or speeding up the instruction fetch rate of specific threads.

Tullsen, et al. [9] investigated several instruction fetch policies for SMT processors. Among their policies, a scheme called *ICOUNT* was found to provided the best performance. The ICOUNT was a priority based approach. Every cycle, ICOUNT gives highest priority to the thread with the fewest instructions in the decode, rename, and instruction queue stages of the pipeline. Thus, ICOUNT prioritizes threads that are likely to make efficient use of processor resources. Nevertheless, the overall instruction throughput from the ICOUNT policy is still significantly lower than the maximum fetch and issue bandwidth of the processor. A primary reason for this is that the ICOUNT scheme is inefficient at reducing the number of wrong-path instructions—that is, incorrectly speculated instructions—in the pipeline. Our measurements show that 17-24% of the instructions fetched by the ICOUNT scheme are from wrong paths. These wrong-path instructions tie up the fetch bandwidth and other valuable resources, such as instruction queues and functional units.

This paper proposes a new fetch scheme that improves an SMT processor's performance by reducing the number of wrong-path instructions in the pipeline. This reduction is achieved by assigning confidence values for the unresolved branch predictions in the pipeline. Once the confidence values for each unresolved prediction are determined, the fetch unit prioritizes threads based on the number of low-confidence unresolved branch predictions they have in the pipeline. Consequently, slowing instruction fetch from a thread with a higher number of low-confidence branches reduces the number of wrong-path instructions in the pipeline. In addition, fetch gating is used to temporarily cut off threads having a large number of low-confidence branches. Experimental results show that our fetch scheme reduces the number of wrong-path instructions fetched to 8-12%, and provides an average performance boost of 14.9%

over ICOUNT.

The rest of this paper is organized as follows. Section 2 reviews background information on simultaneous multithreading and previous work on SMT fetch policies. Section 3 describes our speculation control based fetch scheme. Section 4 presents our experimental results and Section 5 presents the conclusions.

## 2. Background and Motivation

### 2.1. Simultaneous Multithreading Processor

The SMT processing engine, like most other processing engines, comprises two major parts—the fetch engine and the execute engine. The fetch engine is responsible for filling the instruction queues (IQs) with correct-path instructions at a rapid rate, and includes the i-cache, the branch predictor, the fetch unit, the decode unit, and the register rename unit, as shown in Figure 1. The execution engine is responsible for draining the instruction queues at a fast rate, and includes the instruction issue logic, the functional units, the memory hierarchy, the result forward mechanism, and the reorder buffer. As we can see, the important resources are shared by all of the active threads. The key features of the processor can be summarized as follows:

1. All major resources are shared by all of the active threads.
2. Every clock cycle, instructions from all active threads compete for each of the shared resources.
3. Instructions of a thread are fetched and committed strictly in program order.
4. Each of the resources in the processor has very limited buffering capabilities.
5. Once an instruction enters the processor pipeline, it is not pre-empted (i.e., discarded without execution) from the pipeline unless it is found to be from an incorrect path.

Because instructions are not discarded without execution once it enters the pipeline (unless it is determined to be from a wrong path), it is very important to select the right instructions every cycle at the fetch part of the pipeline where instructions enter the pipeline. In this paper, we focus on utilizing the available fetch bandwidth for bringing in the ”best” instructions into the IQs. Effective utilization of the limited resources, particularly the IQs, is very important; in some cycles, the fetch unit may not be able to fetch any correct-path instruction, in which case it may be better to waste the fetch bandwidth rather than to fill up the IQs with wrong-path instructions!

### 2.2. Previously Investigated Fetch Policies

Tullsen et al. had studied fetch policies for SMT processors [9]. In particular, they investigated the following fetch policies, which attempt to improve on the simple round-robin priority policy by using feedback from the processor pipeline.

- **BRCOUNT:** Highest priority is given to those threads that have the fewest unresolved branches in the decode stage, the rename stage, and the instruction queues. The motivation is to reduce the amount of speculation in the processor.
- **MISSCOUNT:** Highest priority is given to those threads that have the fewest outstanding data cache misses. The motivation is that data cache misses take many clock cycles to be serviced, and if subsequent instructions of that thread are data dependent on that value, then they will wait for a long time in the IQs, thereby clogging up the IQs.
- **ICOUNT:** Highest priority is given to those threads that have the fewest instructions in the decode stage, the rename stage, and the instruction queues. The motivation for this policy is two-fold: (i) give highest priority to threads whose instructions are moving through the pipeline efficiently, and (ii) provide an even mix of instructions from the available threads. This naturally prevents any one thread from monopolising the IQs.
- **IQPOSN:** Highest priority is given to those threads whose oldest active instruction is farthest from the head of the IQs. This is based on the assumption that threads with the oldest instructions are most likely to clog the IQs.

All of these fetch policies are quite straightforward to implement. Among these policies, the ICOUNT scheme was found to provide the best throughput in simulation-based evaluations [9]. This is because ICOUNT prioritizes threads that make efficient use of processor resources. For example, a thread with frequent cache misses will frequently stall compared to a thread that has high instruction-level parallelism and no cache misses. The ICOUNT policy will give higher priority to the latter thread and, thereby, boost an SMT processor’s performance. However, it does not take into account if a particular thread is in the correct path of execution or not.

## 3. Fetch Prioritizing and Gating (FPG) based on Confidence Value of Branch Predictions

Because of control speculation, many of the instructions in an SMT pipeline can be from wrong paths. Wrong-path

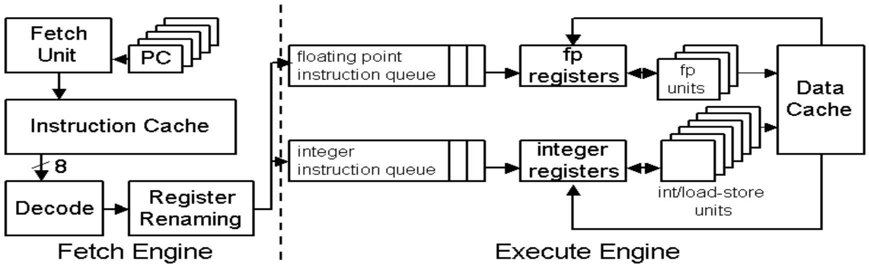


**Figure 1. Block Diagram of an SMT Processor.**

instructions not only do not contribute to the useful instruction throughput, but they also tie up valuable resources, preventing correct-path instructions from being executed. This paper focuses on striving to fill the SMT pipeline with correct-path instructions—that is, with instructions that are from the correct control path—in order to increase the overall throughput of the SMT processor. For SMT processors, speculation control of individual threads is beneficial to improving the overall performance, because the resources that would have been spent on wrong-path instructions of one thread could instead be diverted for use by other threads that are in the right path. To increase the overall performance of SMT processors, we have to reduce the amount of incorrectly speculated instructions, so as to save resources for non-speculative or correctly speculated instructions. An ideal implementation of such a fetch scheme would stop fetching from a thread once it has an outstanding incorrect branch prediction. We call such a scheme as *ideal fetch gating (IFG)*.

### 3.1. Use of Confidence Estimation

In reality, when a processor fetches beyond a conditional branch in a speculative manner, it is not possible to know, at fetch time, if a speculative instruction is from the correct control path or not. Therefore, we use *confidence estimators* [3] [5] to determine which predictions are likely to be correct and which predictions are likely to be incorrect. A high confidence value for a particular branch prediction indicates that the prediction is likely to be correct. A low confidence value indicates that the prediction is likely to be incorrect. Thus, we approximate an ideal fetch gating scheme with the use of confidence estimators to decide which threads to fetch from, and which threads to avoid fetching from, in each cycle.

When classifying predictions into high confidence and low confidence, not all low-confidence predictions may end up being mispredicted, nor will all high-confidence predictions end up being correctly predicted. Each branch prediction, therefore has two attributes:

- its correctness, $\{\texttt{correct}, \texttt{incorrect}\}$
- its confidence, $\{\texttt{high}, \texttt{low}\}$

Strictly speaking, these two attributes are orthogonal, and so four combinations are possible, as depicted in Figure 2. In the figure, $C$ and $I$ denote *correct* predictions and *incorrect* predictions, respectively; and $H$ and $L$ denote *high* confidence and *low* confidence, respectively. It is important to note that a passive confidence estimator (i.e., a setup in which the output of the confidence estimator is not used by the branch predictor to adjust its internal settings) has no control over *C* and *I*; only the branch predictor can modify its settings to change the values of *C* and *I*. The confidence estimator, however, can have some control over *H* and *L*, by varying the threshold value used to categorize a prediction into high confidence or low confidence. The ideal confidence estimator will attempt to minimize both $IH$ and $CL$, while maximizing both *CH* and *IL*. In practice, however, increasing the confidence estimator's threshold to reduce *IH* is likely to increase $CL$, and vice versa. Thus, reducing both $IH$ and $CL$ at the same time is somewhat difficult. The natural question at this stage is: which one is more critical for application in an SMT processor—reducing *IH* or *CL*?

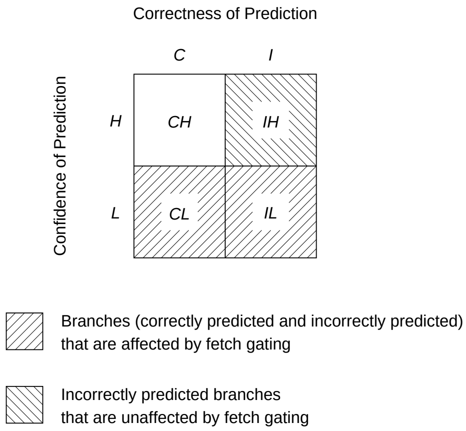


**Figure 2. Classification of Branch Predictions based on Correctness and Confidence**

The answer to the above question depends on the number of active threads that are running on the SMT processor. When the number of active threads is large, there are many threads for the fetch unit to choose from, and so the confidence estimator can be stricter in assigning high confidence to predictions. Although this will increase the number of low-confidence predictions (increasing both *CL* and *IL*), there is still a good chance of having at least one thread with only a few low-confidence predictions. When the number of active threads is small, there are not many threads to choose from, and so it is better for the confidence estimator to be more strict about assigning low confidence to predictions. This is because, if all of the threads have many low-confidence predictions, then no instructions will be fetched at all for a while and the pipeline will be thinly populated.

### 3.2. Fetch Prioritizing and Gating Scheme

Although the use of confidence estimation serves as an approximation to ideal fetch gating, many of the low-confidence predictions signaled by a realistic confidence estimator ends up being correct predictions [4]. In such a situation, if fetch gating is applied to a thread the moment one of its branch predictions ends up having low confidence, and no instructions are fetched from that thread until this branch has been resolved, then the overall performance is likely to be poor. In a particular cycle, if all of the threads have an outstanding low-confidence prediction, then no instructions will be fetched in that cycle. This is despite the fact that many of those threads are likely to be in their correct paths!

In order to deal with the above situation, we propose to allow multiple outstanding low-confidence predictions from each thread, but prioritize the threads based on the number of outstanding low-confidence predictions each thread has. This scheme works well even though the confidence estimator is somewhat inaccurate, because a thread is not cut off just because it has an outstanding low-confidence prediction. In addition to fetch prioritizing, we also investigate the use of fetch gating with larger thresholds; that is, if a thread has a large number of outstanding low-confidence predictions, not only is its fetch priority kept low, but also is it not considered for fetching until one of those predictions is resolved. If the probability for a low-confidence prediction to be wrong is $p$, and we allow up to $n$ outstanding low-confidence predictions, then the probability that at least one of these predictions is incorrect is $1-(1-p)^n$. The maximum number of outstanding low-confidence predictions allowed for a thread is called the *gating threshold*.

The gating threshold should not be too low, because the value of $p$ achieved with most of the confidence estimators is rather low (25%–35%). On the other hand, if the gating threshold is kept high, then it is possible for all active threads to have a large number of unresolved low-confidence branches. In such a scenario, it is better not to fetch from any of the threads. To ensure that, the gating threshold should not be too high either.

The basic idea of our fetch scheme, as we have seen, is speculation control. In other words, we try to reduce low-confidence speculation in the SMT processor. In our fetch scheme, the confidence estimator provides a `high confidence` or `low confidence` value for each branch prediction, based on the branch's past behavior and the confidence estimator's internal threshold value. For each active thread the fetch unit maintains a counter (called *low-confidence prediction counter*) to record the number of unresolved low-confidence predictions in the pipeline from that thread. The priority for each thread is dynamically determined by the value of its low-confidence prediction counter; the highest priority is given to the thread having the smallest value in its low-confidence prediction counter.

Every cycle, the fetch unit will first consider threads with low-confidence prediction counter value zero, and then threads with counter value one, and so on. Priority among threads having the same counter value is determined by the threads' ICOUNT values. Because of taking into consideration the number of low-confidence predictions from each thread, threads with higher confidence in control speculation are likely to engage more resources while those with lower confidence in control speculation will run at an economical rate. The gain from the higher confidence threads will easily surpass the minor loss (if any) from threads with lower confidence, as long as the confidence estimates are accurate. Notice that threads with infrequent branches or highly predictable branches are likely to have higher priority, because they are less likely to have many outstanding low-confidence predictions. The worst case scenario for a thread happens when the predictions for all of its active predictions are marked with low confidence; subsequent instructions from such a thread will not be fetched until some of its branches get resolved. This thread is thus guaranteed to make forward progress, and will not starve.

## 4. Experimental Evaluation

This section presents an experimental evaluation of the fetch prioritizing and gating techniques discussed in Section 3 for improving the instruction throughput of SMT processors.

### 4.1. Evaluation Setup

The experiments in this section are conducted using detailed simulations. Our simulator is derived from the public domain SMT simulator developed by Tullsen et al [9]. The simulator executes unmodified Alpha object code, and models the fetch engine (TLB, branch predictor, fetch unit, decode unit, and register rename unit) and the execution engine (functional units, memory hierarchy, result forwarder, and reorder buffer), along with the instruction queue.

Some of the simulator parameters are fixed as follows: The instruction pipeline has 9 stages, which is based on the Alpha 21264 pipeline, but includes extra cycles for accessing a large register file. Functional unit latencies are also based on 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. Cache line sizes are all 64 bytes. All 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.

Our workload consists of the following 5 programs from the SPEC95 integer benchmark suite: `compress95`, `gcc`, `go`, `li`, and `ijpeg`. These programs have different individual IPC values ranging from 2 to 4, and have different branch misprediction rates. We compiled each program with `gcc` with the `-O4` optimization. The measurement strategy is kept the same as that used by Tullsen et al in [9]: each data point is collected by simulating the SMT processor for a total number of T $\times$ 200 million instructions, where T is the number of threads. We use the following metrics to get a clear idea about the working of our fetch scheme:

- *IPC (Instructions per cycle)*: We measure the overall IPC of the processor, as well as the IPC of each thread.
- *Branch misprediction resolution latency*: This metric measures the average number of cycles a mispredicted branch stays in the pipeline, starting from its fetch time until its execution time.
- *IQ usage / IPC ratio*: This metric is calculated by dividing the average number of IQ slots occupied by a thread (or by all threads) by the IPC delivered by the thread (or by all threads). Notice that lower values for this metric mean better efficiency.
- *Fraction of wrong-path instructions*: This metric indicates the fraction of instructions that are from wrong paths. If we are able to reduce this fraction, then effectively we are doing more useful work in the duration of our measurement.

## 4.2. SMT Configurations Simulated

We simulate the following three SMT configurations:

- **C1:** represents the baseline processor model with 32-slot IQs, 6 integer functional units (4 of them can perform load/store), 3 floating point units, and a fetch bandwidth of 8 instructions per cycle. This is the baseline SMT configuration used by Tullsen et al [9].
- **C2:** This configuration is same as C1, except that the IQs have 64 slots each, and the processor has 12 integer functional units and 6 floating point units.
- **C3:** This configuration is same as C2, except that it has a fetch bandwidth of 16 instructions per cycle.

## 4.3. Fetch Schemes Simulated

**ICOUNT Scheme:** The specific ICOUNT fetch scheme simulated is the `ICOUNT 2.f` scheme from [9]. It fetches up to `f` instructions in a cycle from up to two threads, where `f` is the fetch bandwidth. As many instructions as possible are fetched from the first thread; the second thread is then allowed to use any remaining slots from the remaining fetch bandwidth.

**Fetch Prioritizing and Gating (FPG) Scheme:** We use a JRS confidence estimator [5] to assess the quality of each branch prediction. This estimator parallels the structure of the gshare branch predictor [7]. This estimator uses a table of *miss distance counters (MDCs)* to keep record of branch prediction correctness. Each table entry (MDC) is a saturating resetting counter. Correctly predicted branches increment the corresponding MDC, whereas incorrectly predicted branches reset the MDC to zero. Thus, a high MDC value indicates a higher degree of confidence and a low MDC value indicates a lower degree of confidence. A branch is considered to have high confidence only when the MDC has reached a particular threshold value referred to as the *MDC-threshold*. The default MDC-threshold value is set to 8. The default gating threshold is set to 2; that is, no instructions are fetched from a thread if it has more than 2 unresolved low-confidence branch prediction.

**Ideal Fetch Gating (IFG) Scheme:** We also simulate an *ideal gating* scheme, in which the confidence estimator is perfect, and the gating threshold is 0, that is, this confidence estimator marks as "high confidence" all the correcting predicted branches and marks as "low confidence" all mispredicted branches. The ideal gating scheme is used to study the best possible results with fetch gating.

## 4.4. Results with Single Thread

First we run each benchmark program in single-thread mode in hardware configuration C1 (without any fetch gating) to observe its characteristics. Knowing the characteristic of each program is very helpful in analyzing the multi-thread results presented later in this section. Figure 3 presents the percentage of conditional branches, the branch misprediction ratio, and the average misprediction latency for each benchmark program when it is run in single-thread mode. The misprediction resolution latency is the average number of cycles it takes for a mispredicted branch to get resolved.

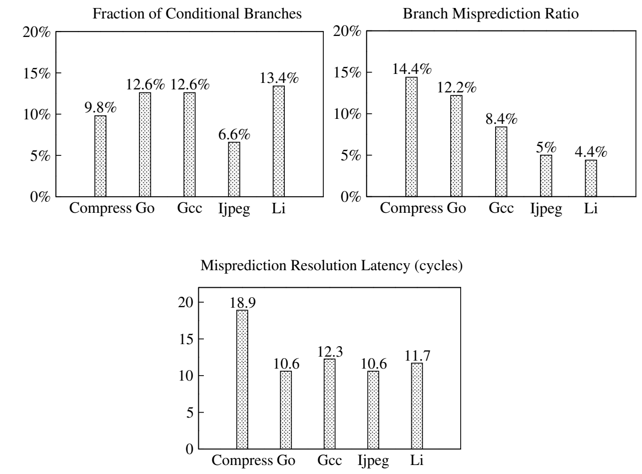


**Figure 3. Performance Characteristics of Benchmark Programs in Single-Thread Mode: (i) Fraction of conditional branches among all instructions; (ii) Branch misprediction ratio; (iii) Misprediction resolution latency**

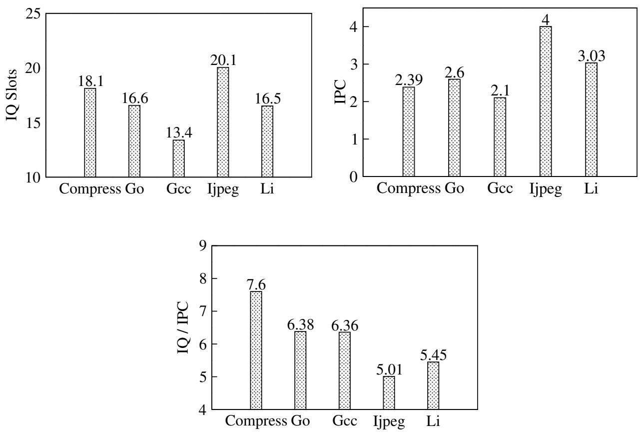


**Figure 4. Performance Characteristics of Benchmark Programs in Single-Thread Mode (i) IQ slots engaged; (ii) Instruction throughput (IPC); (iii) IQ slot occupancy / IPC ratio**

Figure 4 presents the average number of IQ slots occupied, the average instruction throughput (IPC), and the ratio of IQ slot occupancy to IPC. The last parameter indicates the inefficiency of each thread in utilizing the IQ resources. From these figures, we can see that `compress95` takes up more IQ slots than `go` and `li`, for instance, but delivers less IPC than both of them. Looking at Figures 3(ii) and (iii), we can see that `compress95` has the highest misprediction ratio as well as the highest misprediction latency. This means that a large number of wrong-path instructions are fetched for `compress95`, and that they stay in the IQs for a long time (because of the large misprediction latency), without contributing to the IPC. This is clear from Figure 4(iii), where `compress95` has the largest IQ slot occupancy / IPC ratio. A good resource allocation scheme should take unutilized resources away from such programs, and give them to programs that utilize the resources more efficiently.

## 4.5. Results for the FPG Scheme

Next, we simulate the SMT processor in the multithreading mode with the 5 benchmark programs. We measure the total IPC throughput for the three hardware configurations C1, C2, and C3. For each configuration, the IPC is measured using the ICOUNT fetch scheme and our proposed FPG scheme. For the FPG scheme, the MDC threshold is fixed at its default value of 8 (i.e., a branch prediction is deemed to have high confidence if the corresponding MDC value is 8 or more), and the gating threshold is fixed at its default value of 2 (i.e., instruction fetching is stopped from a thread if it has more than 2 unresolved low-confidence branches).

### 4.5.1 Instruction Throughput (IPC)

The IPC results are presented in Figure 5. The figure is divided into three sub-figures, one for each hardware configuration. The first sub-figure corresponds to configuration C1, the second corresponds to C2, and so on. Within each sub-figure, there are 3 groups of histograms, corresponding to IC (ICOUNT), FPG, and IFG (Ideal Fetch Gating) in our discussion. The Y-axis represents the IPC throughput. For each combination of hardware configuration and fetch scheme, 6 histogram bars are presented. The first 5 bars show the IPC contribution of the 5 threads, and the sixth bar shows the overall IPC for that hardware configuration and fetch scheme. Let us analyze the results of Figure 5.

**C1 Configuration:** On comparing the bars for C1-IC and C1-FPG, we can see that the FPG scheme has increased the IPC throughput by $\frac{(5.18-4.45)}{4.45} = 16.4\%$ over the ICOUNT scheme. On comparing the IPCs for the individual threads, we can see that all threads except `compress95` and `go` have obtained higher IPCs with the FPG scheme. Only `compress95`, which had the maximum IQ slot occupancy/IPC ratio, has suffered a decrease in IPC contribution; this decrease is more than offset by the increase in the IPC contributions of the remaining threads. Thus, the

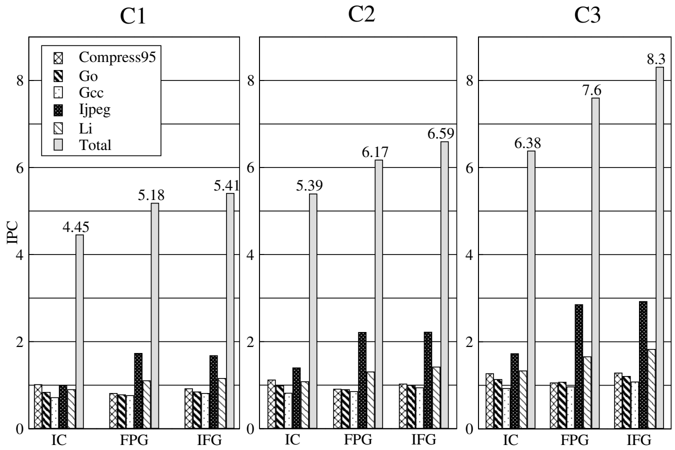


**Figure 5. IPC Throughput comparison of ICOUNT, Fetch Gating, and Ideal Fetch Gating Schemes**

FPG scheme has taken resources away from the "less efficient" `compress95`, and has given them to "more efficient" threads, particularly `ijpeg` and `li`. On comparing the bars for C1-IC and C1-IFG, we can see that the maximum increase in IPC throughput possible by fetch prioritizing and gating is $\frac{(5.41-4.45)}{4.45} = 21.6\%$ over the ICOUNT scheme. Of this, 16.4% has been achieved by our FPG scheme.

**C2 Configuration:** Next, compare the results for the C2 configuration, which uses 64-entry IQs and 12 functional units, but keeps the fetch bandwidth the same. For this configuration, the FPG scheme has obtained a speedup of $\frac{(6.17-5.39)}{5.39} = 14.5\%$ over ICOUNT. This speedup is slightly less than that of configuration C1. This is because, when the IQ size is increased without a corresponding increase in the fetch bandwidth, it becomes less of a critical resource, and so the benefit of fetch gating is less apparent.

**C3 Configuration:** Finally, consider the results for configuration C3, which increases the fetch bandwidth to 16 instructions per cycle. With the larger fetch bandwidth, it becomes more important to use a good fetch scheme, otherwise the IQs get filled with incorrect instructions sooner! For configuration C3, the FPG scheme obtains a speedup of $\frac{(7.60-6.38)}{6.38} = 19.1\%$ over ICOUNT.

#### 4.5.2 IQ Usage / IPC Ratio

In order to throw more light on the IPC results reported above, we next present the IQ usage / IPC ratio values. which indicates the inefficiency of IQ usage. The metric is measured for each thread as well as for the 5-thread aggregate. These results are presented in Figure 6. The configurations and format for this figure are the same as in Figure 5.

On comparing the histogram bars for the ICOUNT and FPG schemes, we can see that the latter is able to better utilize the IQ resources. For instance, in configuration C3, when ICOUNT-based fetching is employed, `compress95` suffers from a large IQ usage / IPC ratio which leads to poor utilization of the IQ slots. The FPG scheme, on the other hand, has taken resources away from `compress95`, and has diverted it to other threads that make better utilization of the hardware resources.

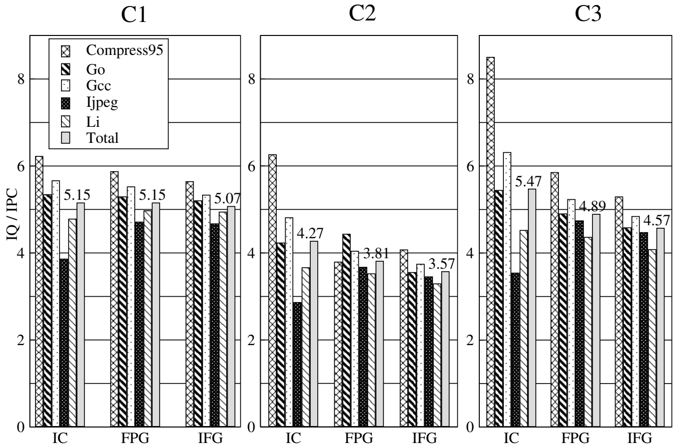


**Figure 6. Average IQ Usage / IPC Ratio**

#### 4.5.3 Wrong-Path Instructions

It is also illuminating to measure the percentage of wrong-path instructions fetched with the ICOUNT and FPG schemes. Figure 7 shows the percentage of wrong-path instructions in the pipeline under these schemes for the C1, C2, and C3 configurations. For each combination of configuration and fetch scheme, two histogram bars are shown. The first bar shows the percentage of fetched instructions that belong to wrong paths, and the second bar shows the percentage of executed instructions that belong to wrong paths.

For the C1 configuration, compared to the ICOUNT scheme, the FPG scheme reduces the percentage of fetched instructions belonging to wrong paths from 17.0% to 9.0%, and the percentage of executed instructions belonging to wrong paths from 9.0% to 4.6%. Reducing the percentage of wrong-path instructions in the instruction pipeline leads to better utilization of the pipeline, which translates to better IPC throughput, as we saw earlier.

## 5 Summary and Conclusions

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

This paper proposed a new instruction fetch scheme called *fetch prioritizing and gating* (FPG) for SMT processors. The basic idea is to allow aggressive speculative execution for threads that have high branch prediction rates while limiting speculation on threads with lower prediction rates. This scheme sets up fetch priority for each thread using the number of unresolved low-confidence branches from the thread. Based on the fetch priority of each thread, the fetch scheme finds threads that are most likely to be in their correct paths. By limiting the amount of low-confidence control speculation applied to a particular thread, resources can be better distributed to achieve higher throughput. Our experimental evaluation showed that this fetch scheme provides up to 17.4% speedup over ICOUNT, which is the best fetch policy reported so far for SMT.

We expect the advantage of our fetch scheme to be more prominent with a deeper pipeline with a larger branch misprediction penalty and with higher degrees of multithreading. Fetch prioritizing and gating should help longer pipelines because it reduces the number of wrong-path instructions by as much as 50%. This scheme should help higher degree of multithreading because more threads may compete for fewer resources, which makes it critical to fetch instructions from the thread that has the fewest wrong-path instructions.

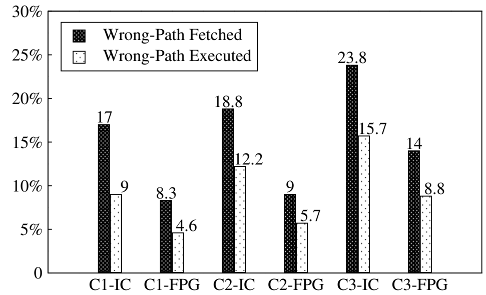


**Figure 7. Percentage of fetched instructions and percentage of executed instructions belonging to wrong paths**

## Acknowledgements

This work was supported by the U.S. National Science Foundation (NSF) through a regular grant (CCR 9711566) and a CAREER grant (MIP 9702569).

## 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 Annual International Symposium on Computer Architecture*, 1998.

[4] S. Manne, A. Klauser, and D. Grunwald, "Pipeline Gating: Speculation Control for Energy Reduction," *Proc. 25th Annual International Symposium on Computer Architecture*, 1998.

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

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

[8] 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.

[9] 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 Annual International Symposium on Computer Architecture*, pp. 191-202, May 1996.

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