The Impact of Fetch Rate and Reorder Buffer Size on Speculative Pre-Execution*

David M. Koppelman

Department of Electrical & Computer Engineering , Louisiana State University

koppel@ece.lsu.edu

Abstract

Pre-execution systems reduce the impact of cache misses and branch mispredictions by forking a slice, a code fragment derived from the program, in advance of frequently mispredicted branches and frequently missing loads in order to either resolve the branch or prefetch the load. Because unnecessary instructions are omitted the slice reaches the branch or load before the main thread does, for loads this time margin can reduce or even eliminate cache miss delay.

Published results have shown significant improvements for some benchmarks, on the order of 20%, with many showing at least single-digit improvements. These studies left unexamined two system parameters that one would expect pre-execution to be sensitive to: fetch rate and reorder buffer size. Higher fetch rate would allow the main thread to reach the troublesome load sooner, but would not affect the slice and so the slice's margin is reduced. Studies have shown large potential margins for slices, but the fetch rate effect has not been measured. A second system parameter is reorder buffer size. A larger reorder buffer would allow a system to hide more of the miss latency that pre-execution reduces.

To test the sensitivity to these factors pre-execution schemes were simulated on systems with varying fetch rates and reorder buffer sizes. Results show that higher fetch rate does not reduce pre-execution speedup in most benchmarks. Reorder buffer size sensitivity varies, some benchmarks are insensitive to reorder buffer size increases beyond 256 entries, but still benefit from pre-execution, the benefit due in large part to prefetching those loads that provide values for frequently mispredicted branches. The benchmarks that are sensitive to reorder buffer size are also the ones that benefit most from pre-execution.

1. Introduction

Pre-execution schemes are one approach to reducing the impact of cache misses and branch mispredictions. In a pre-execution scheme troublesome (frequently missing) loads are identified and for each troublesome load a *slice* consisting of the load and instructions producing its address is constructed and cached. For each slice a *trigger* instruction is identified, the next time the trigger is encountered the slice will be retrieved and executed. If successful the load in the slice, called the *prefetch load*, will execute several cycles before the load for which the slice was constructed, reducing or eliminating cache miss latency. Slices do not affect state visible to the running program and so they may be killed at any time, only at the cost of missing a prefetch opportunity. Slices can also be constructed for troublesome branches, though with the added complication of matching the predicted outcome to the main-thread instruction stream.

In some schemes slices are constructed dynamically and in hardware, in others they are prepared in software based on a profile run. The slice might be a subset of the dynamic instruction stream, or it might be optimized in some way. Slice decode and execution share existing processor resources in some schemes, in others slice execution uses its own decode hardware, execution hardware, or both. For more details see Section 2.

What is common in the pre-execution schemes examined here is that the slice consists of ordinary machine instructions which make use of processor state from the running program, in particular register values, and that the slice is forked when the program reaches some instruction.

The use of machine state and execution resources for slice execution is costly in one way or another. If execution resources are duplicated the cost is direct, and in all schemes at least the register map or register

* Presented at the Workshop on Duplicating, Deconstructing, and Debunking, held in conjunction with the 30th Annual International Symposium on Computer Architecture, June 2003. Presentation slides available via http://www.ece.lsu.edu/koppel/pubs/pe_wddd_pr.pdf.

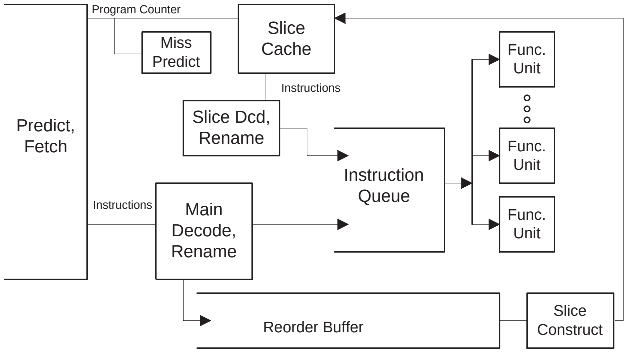

Figure 1. Typical pre-execution hardware organization. Slice instructions share functional units but not reorder buffer slots. Illustrated (and simulated) system has separate decode/rename for slices. In other proposed schemes slice shared decode/rename with the main thread.

values must be copied. If resources are shared then instruction queues and other scheduling hardware must be enlarged. When implemented on top of simultaneous multithreading hardware, slices displace ordinary threads.

These costs are offset by the potential benefit, which there is plenty of since load latency is no small problem. Zilles and Sohi [21] show that performance would improve by a large amount, more than doubling for some SPEC2000 integer benchmarks, if troublesome loads hit the cache and troublesome branches were correctly predicted. Pre-execution can prefetch loads which conventional hardware prefetch schemes cannot, such as those generating irregular address patterns.

The various pre-execution schemes described in the literature realize respectable, in some cases large, speedups. For example, Collins, Tullsen, Wang, and Shen demonstrate an average speedup of over 1.3 on memory-intensive benchmarks.

The published analyses of pre-execution schemes looked at factors such as slice construction techniques, but for the most part evaluated pre-execution on a single type of system [1,3,10,11,14,21]. There are two system parameters to which pre-execution may be sensitive: the fetch rate and reorder buffer size. A higher fetch rate will reduce the time advantage of a given slice, or require triggering a slice further back, risking triggering a slice for the wrong path. Reorder buffer size is an important factor because a larger reorder buffer can hide the load miss latency that pre-execution reduces. (Put another way, pre-execution can reduce the latency that would otherwise require a larger reorder buffer to hide.) A related advantage of smaller reorder buffers for pre-execution is that they are more frequently full. That, of course, stops the main thread but not a slice that has been triggered.

The impact of fetch rate and reorder buffer size on pre-execution schemes will be examined here. A pre-execution scheme will be simulated on systems having varying fetch rate, reorder buffer sizes, and slice construction window sizes. The impacts on performance, and the reason for that impact will be analyzed in detail.

The remainder of this paper is organized as follows. A discussion of some existing pre-execution schemes appears in the next section. Pre-execution performance factors are discussed in Section 3. Details of the simulated system and benchmarks are described in Section 4. Experiments are described and discussed in Section 5, related work is discussed in Section 6, and conclusions appear in Section 7.

2. Pre-Execution Schemes

Perhaps triggered by Zilles and Sohi's 2000 study [20] a number of pre-execution schemes had been published in 2001. These will be discussed here, while antecedents and other related schemes are discussed Section 6. An outline of pre-execution and related terminology is presented below, followed by details from published studies and the version simulated here.

2.1. Basic Pre-Execution and Terminology

In a pre-execution scheme *troublesome* loads are identified, these are loads which miss the cache and in some variations are believed to be on a critical path. The instructions preceding a load needed to compute its address are said to be in its *dataflow tree*; the number of preceding instructions considered is called the *construction window size*. When a troublesome load is identified its dataflow tree is constructed, and in some variations optimized; the result is called a *slice*. The slice is placed in a *slice cache*. With a slice constructed for it, the troublesome load is known as a *target* and the corresponding instruction in the slice is called the *prefetch load* (briefly, prefetch). A *trigger instruction* (briefly, trigger) is selected, often the earliest instruction in the construction window. The next time fetch (or more often, a decode step) reaches the trigger the slice is retrieved and *forked*, that is, its execution is started, and will proceed in parallel with the *main thread*. Depending on variation the forked slice may share decode/rename resources and execution resources. The prefetch does not wait if it misses the cache.

2.2. Published Pre-Execution Schemes

Some of the schemes discussed below use pre-execution for branches as well as loads; only the portions handling loads are discussed.

Finding troublesome loads may be harder than it sounds, at least for a run-time mechanism. Only two schemes identify troublesome loads at runtime, Dynamic Speculative Precomputation (DSP) of Collins, Tullsen, Wang, and Shen [3] and the Slice Processor (SP) of Moshovos, Pnevmatikatos, and Baniasadi [11]. The SP uses a load miss predictor to select target loads (on a miss add 4 to a counter in a PC-indexed table, on a hit subtract 1), the DSP goes further by finding only critical-path loads (based on the time spent waiting at the head of the reorder buffer). The other schemes mentioned here rely on some form of profiling to select target loads.

There are major differences in the way the slice itself is constructed. Unmodified dataflow trees are used by SP, the Dependence Graph Precomputation (DGP) scheme of Annavaram, Patel, and Davidson [1], and speculative Data Driven Multithreading (DDM) of Roth and Sohi [14]. Tree construction occurs dynamically (at run time) in SP and DGP; in SP the tree is constructed from the last 32 (nominally) committed instructions while in DGP it is constructed from instructions waiting in the fetch queue. In DDM tree construction occurs in a pre-processing step.

There are many opportunities to optimize instructions in the dataflow tree, for example replacing or eliminating store/load pairs, and combining arithmetic instructions (say, two instructions incrementing the same variable). The DSP optimizes slices dynamically while Zilles and Sohi [21] in what will be called the Speculative Slice Study (SSS) here, construct their slices by hand.

A refined approach to slice selection and construction, called the Quantitative Framework (QF) here, is presented by Roth and Sohi [15]. A score is constructed for a candidate slice using the margin of the prefetch over the target load adjusted for the number of times the target load will be reached, the score also includes the number of instructions in the slice (which reduces the score). Slices are selected from a set of candidates based on how well they cover target loads, combining the effect of a short, low-margin slice against a longer, high-margin slice that is less likely to reach the target.

In some cases the slices can contain loops. When constructing slices DSP hardware specifically looks for a second instance of the troublesome load so that the constructed slice can have a loop. A loop counter and call level monitor terminate slices that may loop too long. The hand-constructed slices in the SSS also contain loops and use a maximum iteration counter.

The various pre-execution schemes differ in the resources they use to execute slices. In the boldest of these schemes slices execute as a special thread on a simultaneous multithreaded (SMT) [8] machine, competing for decode/rename and execution resources with the main thread. With this resource sharing there is significant potential for slowing down the main thread. SMT (or SMT-like) execution is done by the DSP, DDMT, and SSS. DSP and DDMT do not compete for reorder buffer (ROB) slots with the main thread; it is not clear whether this is true in SSS. An alternative is to provide separate decode/rename resources for slices, this is done in the SP. The DGP is something of a special case since it operates on instructions in the normal fetch stream, relying on run-ahead of the fetch unit.

Of all the schemes only the DGP provides its own execution resources, in the others slice instructions compete with the main thread for functional units, in some cases at a lower priority.

2.3. The Expensive Slice Machine

A goal of this study is to analyze the impact of fetch rate and ROB size on pre-execution performance, not to find a good cost/performance balance. Therefore a pre-execution scheme was chosen for analysis that gives good performance with little regard for resources used. That is, the size of the slice cache and the number of functional units is large, so as to bring out as much potential performance as possible; it will be called the Expensive Slice Machine (ESM).

Slice construction is performed as loads commit; it is performed instantaneously and for all committing loads generating a *lead miss*. ESM constructs (but does not always cache) slices for every load instruction that generates a lead miss. A lead miss is the miss to the level 1 cache that initiates a cache fill. (A following instruction that access the same address before the data arrives is not said to generate a lead miss.) The slice is constructed by extracting the dataflow tree for the troublesome load using a buffer holding recently committed instructions. The *margin* for a newly constructed slice is computed, the number of cycles by which the prefetch load will precede the target load assuming that when the slice is forked fetch proceeds at the same rate as when it was constructed. If the margin is below 2 cycles the slice is not cached.

A *miss distance* is stored with each slice, this is used to determine whether to fork the slice. When the slice is constructed the miss distance is set to zero, it is also set to zero if the slice's prefetch instruction generates a lead miss; it is incremented if the prefetch does not generate a lead miss. (Any practical system would have a separate load miss predictor.)

The number of simultaneously executing slices is limited to eight. If less than eight are executing the slice cache is probed using the addresses of decoding instructions. If a slice is found and its miss distance is less than 7, execution forks. The slice is decoded and renamed at a rate of 4 instructions per cycle in the base machine. Slice instruction execution proceeds normally, that is, decode, rename, and scheduling proceed at the same speed as for ordinary instructions.

Slice instructions do not use ROB slots but they share execution resources of main-thread instructions. In the systems analyzed there are plenty of functional units and so there is rarely any waiting.

3. Pre-Execution Performance

The speedup attained by pre-execution depends in part on the margin of the prefetch over the target and by the number of targets that lie on a critical path.

3.1. Critical Path and Sensitivity to ROB Size

In a dynamically scheduled system some load misses do not impact execution time, instead the processor "catches up" when these loads complete. Those that do impact execution time in a particular processor configuration are called *critical loads* for that configuration.

There are two ways loads can be critical: they can delay branch or jump target resolution, or they can contribute to the filling of the reorder buffer. The first type will be called *control-critical loads*, the second type will be called *window-critical loads*. Control-critical loads are those loads that, because they don't hit the cache, force instructions computing a branch condition or target for a mispredicted branch to wait. For window-critical loads if the load had not missed the reorder buffer would have filled later, if at all. In both cases the load delays correct-path instruction fetch.

Pre-execution will reduce the miss delay of both types of critical loads, the data presented below shows these effects separately. Reducing the impact of control-critical loads is something that is difficult to do by means other than larger or faster caches or better branch predictors. In contrast, the impact of some (perhaps most) window-critical loads can be reduced by increasing the size of the reorder buffer. For that reason, reorder buffer size sensitivity is important, and is being investigated here.

Apart from just pre-fetching loads early, another way that pre-execution, at least the published schemes, improves performance is by allowing a slice to be fetched and executed when the rest of the system is stalled due to a full reorder buffer. This increases the margin when the target load is caught waiting outside a full reorder buffer while the pre-fetch executes. (In most of the previous studies slices do not share reorder buffer slots with the main thread.)

3.2. Sensitivity to Fetch Rate

As used here the *fetch rate* is the number of decoded instructions (including those that will be squashed) divided by the number of cycles at which decode was not stalled (due to a full reorder buffer, lack of physical registers, etc). The fetch rate is determined by decode width and by the front end, the part of the processor that predicts and fetches instructions. A processor with a decode width of 8, that is, an 8-way superscalar processor, has an ideal fetch rate of 8 instructions per cycle. Due to limitations of the instruction cache and the ability of the front end to predict multiple control transfers per cycle, the front end may deliver less than eight instructions per cycle. Advanced front ends, such as trace caches [13] and multiple branch predictors [19] have higher fetch rates, approaching the maximum possible (the decode width).

Assuming no stalls around the time a slice is forked, the margin for the slice is determined by the fetch rate, the trigger-to-target distance, and the dataflow distance. That there is plenty of margin for troublesome loads was revealed in the study of Zilles and Sohi [20]. They identify troublesome loads, in particular those that have the most impact on overall execution, and plot the number of instructions in the dataflow tree versus the distance from the target load. They show that from 5% to 30% or more of instructions preceding a load are in its dataflow tree, the lower number assuming all stores can be omitted and the higher including all stores. That this margin could be exploited was demonstrated in the many projects described above.

One important question unanswered by these studies is the degree of sensitivity to fetch rate. In most of these studies some estimate is made of margin and slice candidates not meeting this margin are rejected. A higher fetch rate would mean more slice candidates would be rejected (or would be ineffective), and so reduce the impact on performance. Based on observed distribution of instructions [20] moving the trigger back would increase the margin, but at the risk of placing the trigger before a frequently mispredicted branch. Only the QF takes this slice survivability into account [15].

4. Evaluation

4.1. Simulator

The systems were analyzed using RSIM [12], a detailed microarchitecture simulator. Modifications were made to simulate pre-execution and many other unrelated modifications were made. RSIM is a microarchitecture simulator which simulates a dynamically scheduled superscalar processor and memory system. The processor implements a subset of the SPARC V8 ISA

Table 1. Benchmarks. (Table numbers in millions.)

Bench- mark	Insn Skip'd	Insn Sim'ed	Loads Cmtd	L2 Hits	L2 Misses
mst	1200	500	78	2.34	18.88
em3d	120	300	35	0.33	0.40
bzip2	0	305	78	9.61	3.00
gcc	0	607	108	5.21	0.26
gzip	0	636	108	26.53	1.86
mcf	1000	100	27	2.07	12.83
perl	0	181	35	2.59	0.07
swim	0	400	104	9.85	44.11
TeX	0	102	20	0.58	0.03
vpr	5000	300	69	4.74	3.87
wupwise	5000	500	170	0.28	0.95

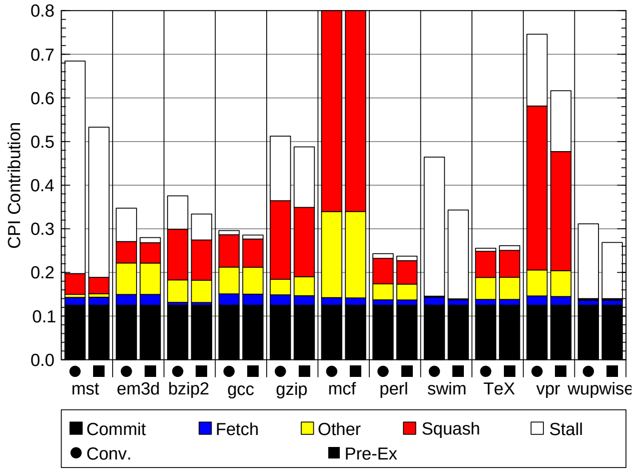

Figure 2. Performance of a conventional system and one using pre-execution. Segments show CPI contribution.

[18]. Benchmark programs are compiled exactly as they are for a real system. Linking is identical except for the use of static libraries (though still the system's libraries, not specially prepared versions) and a special startup file. System calls are not simulated.

Dynamic execution is aggressive: The register map used for renaming is checkpointed when branches or jumps are decoded so that recovery can start when mispredicted instructions resolve. Exception recovery is initiated when the faulting instruction is ready to commit. Other system characteristics are summarized below:

Front End

Branch target prediction using a basic block predictor [19] with a 2^{15}-node block address cache; directions predicted with a variation on a YAGS predictor[4] using a 16-bit GHR. Base system has three instruction cache ports and predicts two blocks per cycle. One- and two-icache-port systems predict one block per cycle; four-port system predicts three blocks per cycle. Block predictions are queued. Indirect jump predictor uses a 2^{16} entry GHR-indexed table. Returns predicted with an 8-entry RAS.

Core

Base system 8-way superscalar; three cycle delay from decode to earliest execution opportunity. Reorder buffer holds 256 entries; virtually unlimited functional units. Instruction queues and load/store queues have virtually unlimited space.

Memory

L1 instruction cache: 328 kB, 5-way. L1 data cache: 16 kiB, 4-way, 64-byte line; 2-cycle hit latency including address generation. L2 data cache: 256 kiB, 8-way, 16-cycle hit latency. Memory, 100 cycle access latency plus congestion and overhead.

Pre-Execution

Slices constructed from 256-instruction window; maximum slice size 64 instructions; slice rejected if margin (prefetch to target) less than 2 cycles. Slice cache size 2^{16} entries. Slice not forked if target load has more than 6 consecutive non lead misses (a hit or miss to a line already on the way). At most 8 slices in flight; slices injected at half the decode width (4 instructions per cycle in base system). Slices execute with shared execution resources, but use private decode and rename and do not use reorder buffer slots.

4.2. Benchmark Programs

The simulated programs come from the SPEC and Olden suites. Except for vpr, the SPEC programs were not selected for pre-execution suitability. (That is, no program was selected because of favorable speedup or any other performance reason.) The Olden benchmarks are a set of pointer-intensive microbenchmarks adapted by Luk [9] for use in studying prefetching and used later by other investigators to test the effectiveness of pre-execution and other schemes. They are included here so results can be compared to other studies that use these benchmarks.

Benchmarks vpr, swim, gzip, mcf, and wupwise are compiled using the SPEC CPU2000 makefiles, using code from that suite. The code for the other benchmarks was obtained from their standard distributions, compiled with optimization. Optimization was targeted to an UltraSPARC II processor, so scheduling would not perfectly match the wider-issue systems simulated here.

Benchmark swim uses test inputs, mcf and wupwise use reference inputs. Benchmark vpr uses reference inputs but is only run for placement. Olden benchmark em3d uses input 25000 100 75 1, benchmark mst uses input 3407 1. The other spec benchmarks use shortened inputs. Table 1 summarizes benchmark characteristics including the portion of the benchmark simulated.

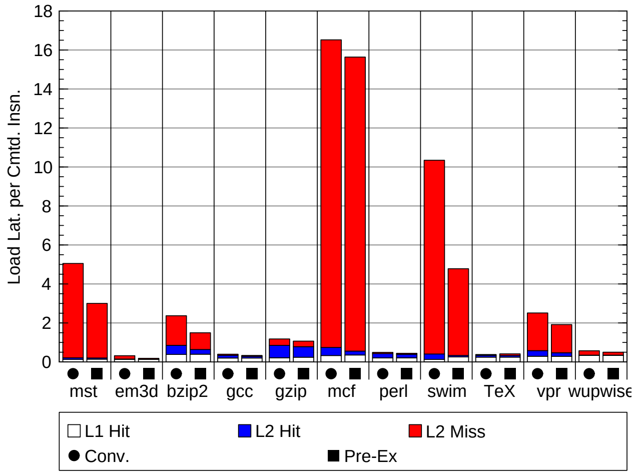

Figure 3. Load latency per committed instruction of a conventional system and one using pre-execution. Segments show contribution from level 1 hits, level 2 hits, and misses.

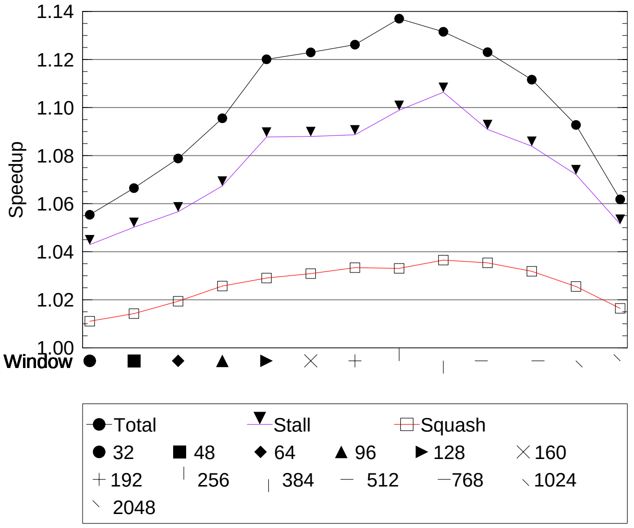

Figure 4. Speedup of pre-execution with varying construction window sizes. Total is the actual speedup, Stall shows the speedup due to window-critical loads, and Squash shows the speedup due to control-critical loads.

5. Experiments

The performance of conventional and pre-execution systems is shown in Figure 2. The total height of each bar is the execution time in cycles per instruction (CPI), the segments show the ideal execution time, **Commit**, and four factors degrading performance: correct-path stalls, **Stall**, misprediction squashes and wrong-path stalls **Squash**, instruction cache port limitations **Fetch**, and miscellaneous factors, **Other**. Control-critical loads primarily effect the **Squash** segment while window-critical loads primarily effect the **Stall** segment.

The size of each segment is determined by tallying how each decode slot is used at each cycle and dividing each tally by the number of committed instructions times the decode width. The **Commit** segment shows slots used by committed instructions, its height is the ideal execution time of $\frac{1}{8}$ CPI. The **Fetch** segment shows decode slots unused due to a limit on the number of instruction cache ports. The **Squash** segment shows slots wasted due to mispredicted or unpredicted control transfers. This includes slots holding instructions that survive long enough to be scheduled but are ultimately squashed due to mispredictions and slots empty because the ROB is full while waiting to fetch down a mispredicted path. The **Stall** segment shows slots empty because the ROB is full while waiting to fetch down a correct path. The **Other** segment shows other cases. For most benchmarks this is dominated by instructions that are squashed before they could be scheduled and instruction cache misses.

The benchmarks show variation in their CPI, the performance loss due to control- and window-critical loads, and in how much pre-execution helps. Benchmarks perl, TeX, and gcc are the most efficient and are little improved by pre-execution while most of the less efficient benchmarks enjoy more substantial speedup with the exception of gzip, which is ILP limited.

The benchmarks' sensitivity to pre-execution is determined in part to the number of cache misses. Load latency per committed instruction is plotted in Figure 3. The segments show the contribution of loads that hit the level 1 cache, hit the level 2 cache, and those that miss the level 2 cache. Benchmark mcf has the longest load latency but because the loads that miss the cache are participating in a long pointer chase pre-execution can do little to reduce the latency.

5.1. Speedup and Cons. Window Size

Average speedup is plotted in Figure 4 and the speedup of selected benchmarks is plotted in Figure 5. In both plots speedup is shown for pre-execution schemes with construction windows varying from 32 to 2048 instructions. The points marked **Total** show the speedup over the conventional system. Benchmarks vary in the size of the window needed (based on the level 2 cache hit ratio), average performance peaks at a window size of 384. Performance drops with further increase in window size because of the limit of eight in-flight slices. The base construction window size of 256 achieves close to maximum performance.

In addition to total speedup these figures show an estimate of the speedup obtained by improving only control-critical loads, **Squash**, and window-critical loads, **Stall**. The control-critical speedup estimate is

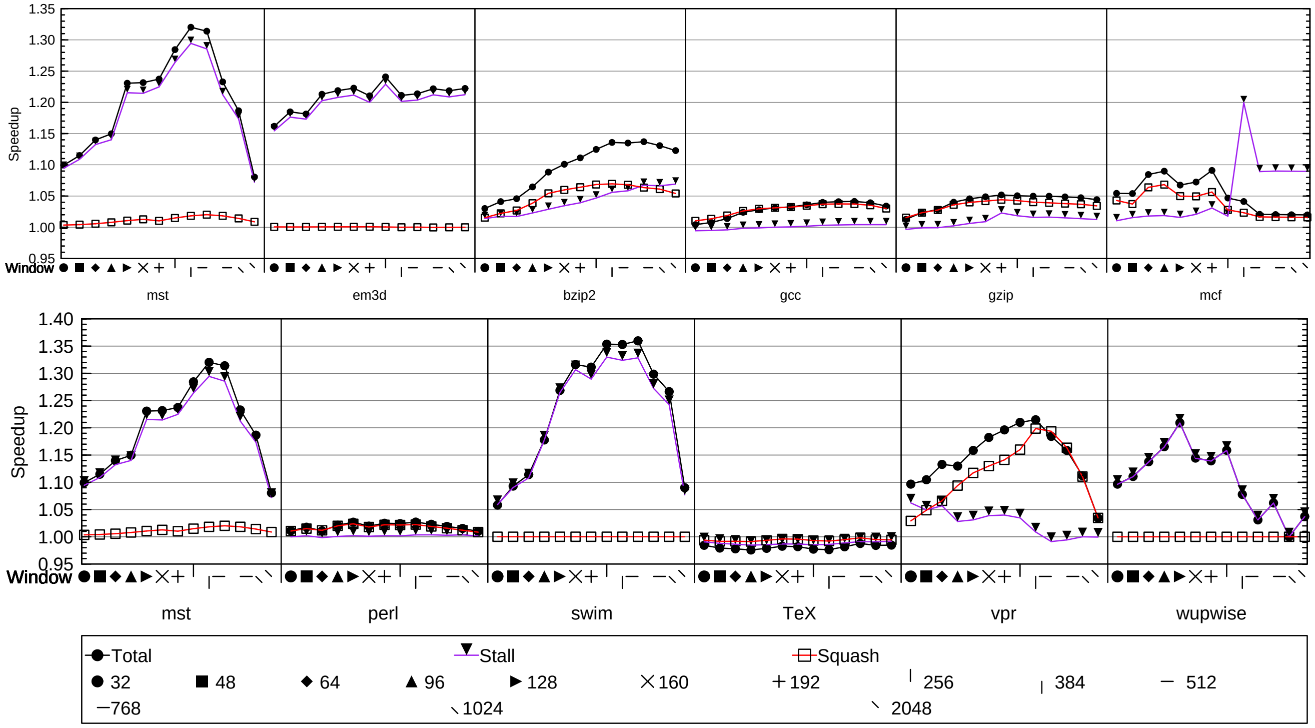

Figure 5. Speedup of pre-execution with varying construction window sizes.

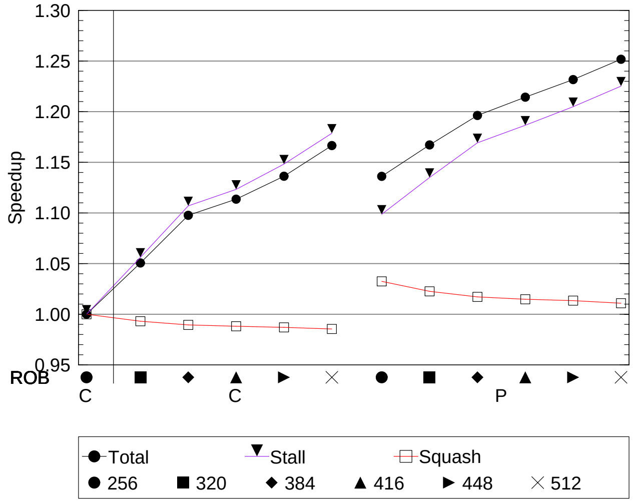

Figure 6. Average speedup of benchmarks on systems with, P, and without, C, pre-execution with varying reorder buffer sizes. Speedup is over conventional system with a 256-entry ROB.

obtained by essentially replacing the **Squash** segments (from Figure 2) of the conventional system with the corresponding segments in the pre-execution system and comparing the result to the unmodified conventional system. A similar approach is used for the window-critical speedup.

From Figure 4 one can see that for these benchmarks most performance improvement is from window-critical loads. Looking at individual benchmarks in Figure 5 one can see that for some benchmarks, such as mst and swim, almost all improvement is for window-critical loads, for others control critical loads are more important.

5.2. Reorder Buffer Size

The impact of reorder buffer size and pre-execution on critical loads is shown in Figure 6 where speedups are plotted for systems with reorder buffer sizes from 256 to 512, the left group is conventional, the right is using pre-execution. All speedups are with respect to the base system (with a 256-entry ROB).

On average the speedup obtained with pre-execution and a 256-entry ROB is about the same as a conventional system with a 448-entry ROB. If both systems are feasible, the less expensive system is better. (Further below larger ROB systems also have longer pipelines.)

As expected, both pre-execution and larger ROBs reduce the impact of window-critical loads. Comparing a 512-entry ROB conventional system to a 256-entry pre-execution system, the two center points, shows better performance on the conventional system. The pre-execution system is not as effective at improving window-critical loads but unlike the conventional system can improve control-critical loads.

Pre-execution is still effective on systems with larger

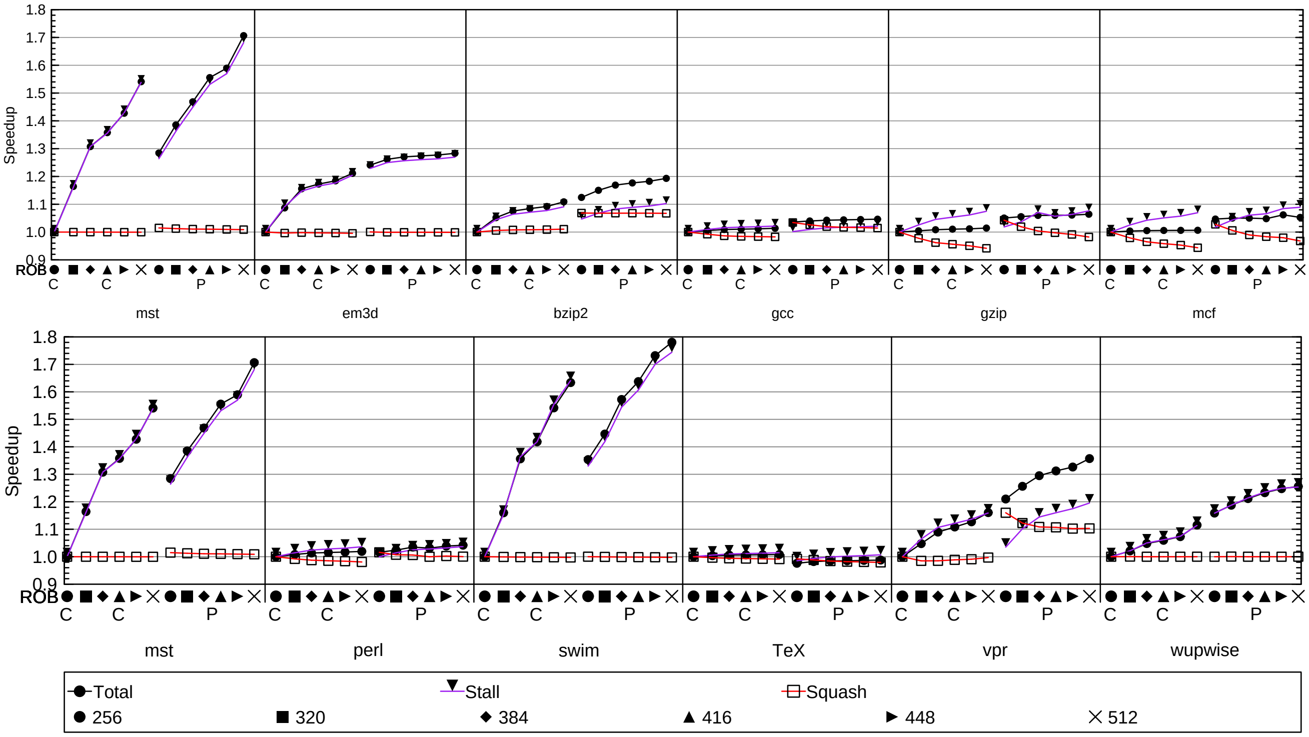

Figure 7. Speedup of selected benchmarks on systems with, P, and without, C, pre-execution with varying reorder buffer sizes. Speedup is over conventional system with a 256-entry ROB.

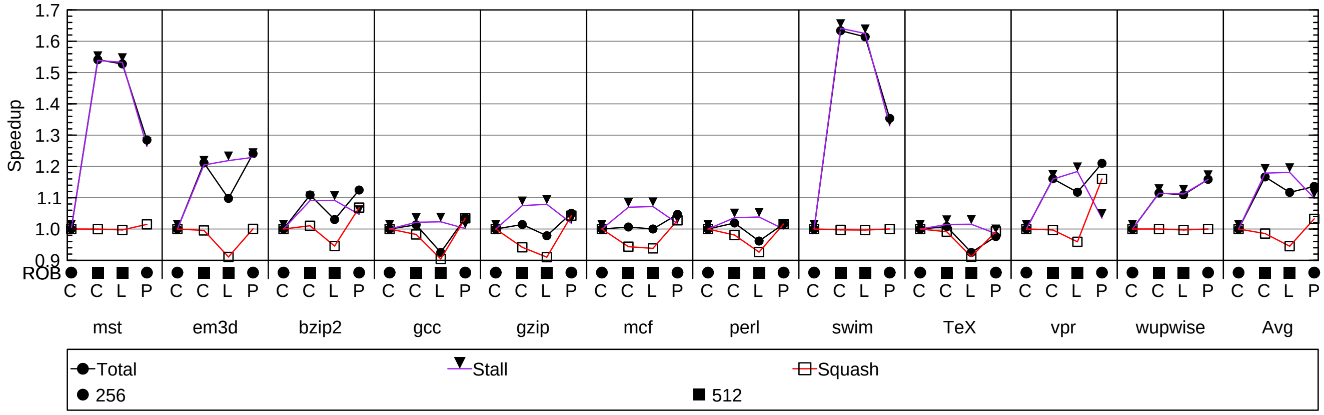

Figure 8. Speedup of a conventional system with a larger ROB, second C, one with a larger ROB but with 3 additional scheduling pipeline stages, L, and a pre-execution system with a 256-entry ROB, P.

ROB sizes, at least up to 512 entries. On the base systems it would take more than 800 entries to cover the level 2 miss latency.

Figure 7 shows the same data for selected benchmarks. Pre-execution is able to out-perform a larger ROB on bzip2, gzip, and vpr due to the control-critical loads that can be helped.

In the comparisons above the systems with a larger reorder buffer got it for "free," in real systems ROB size may be limited by critical paths in scheduling queues needed to hold pending instructions. Figure 8 shows the speedup of three systems: a conventional system with a 512-entry ROB (the same as the one used above), a conventional system with a 512-entry ROB and three additional stages in the scheduling pipeline, **L**, and a pre-execution system using a 256-entry ROB.

The longer scheduling pipeline is felt after branch mispredictions, its impact can be seen in the **Squash** speedup component. For four of the benchmarks this results in a slowdown over the base system, for the others it reduces the speedup over the conventional system with the unmodified pipeline.

When compared to a conventional system with a

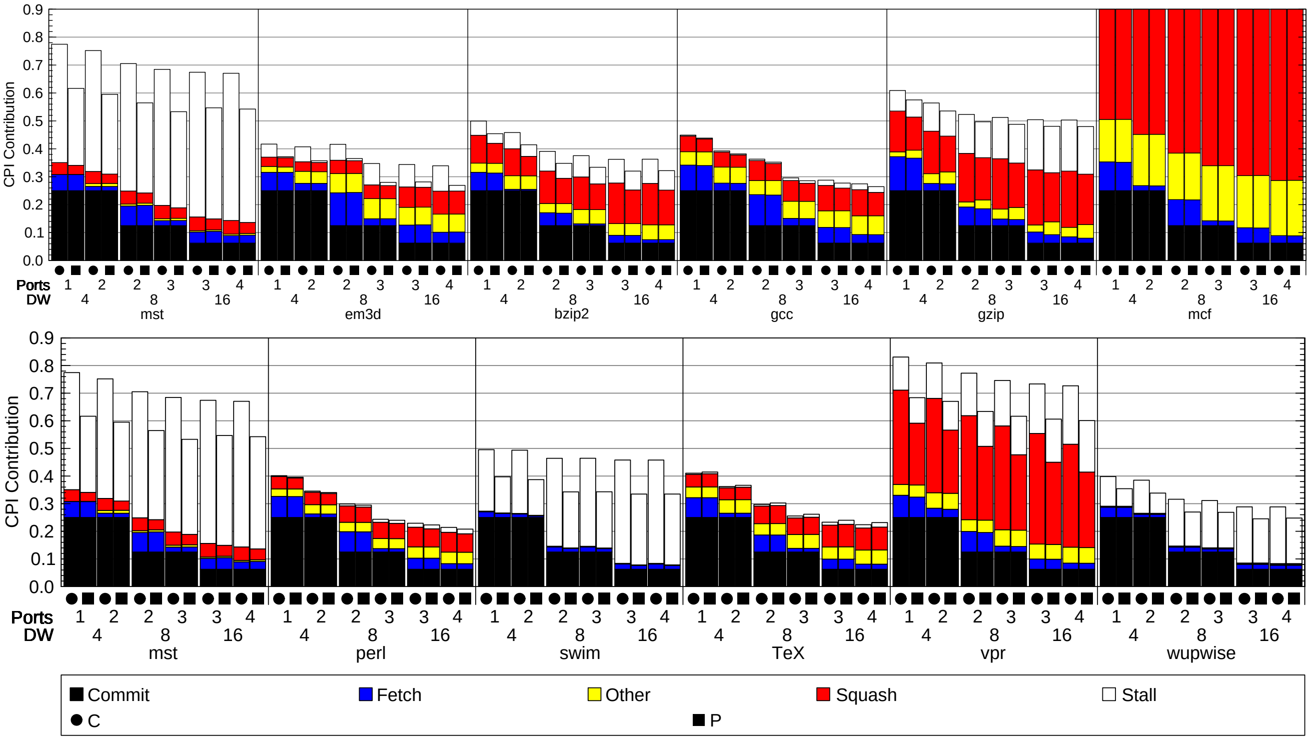

Figure 9. Performance of conventional and pre-execution systems with different front ends.

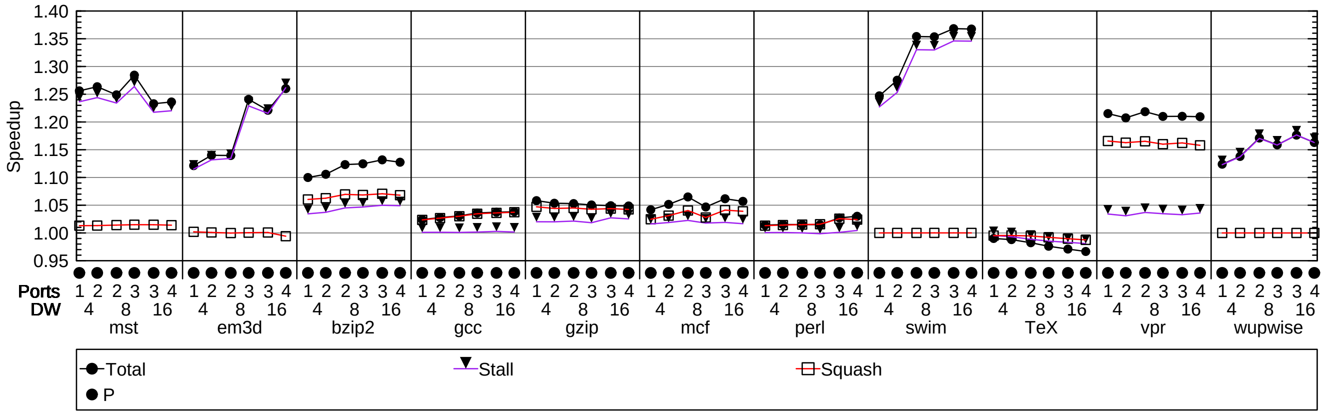

Figure 10. Speedup and speedup contributions of conventional and pre-execution systems with different front ends.

longer pipeline pre-execution does much better, outperforming the conventional system on all but two benchmarks (one of which is synthetic).

5.3. Fetch Rate

Fetch rate is another system parameter that can affect pre-execution. Systems with varying front ends were simulated, including 4-, 8-, and 16-way processors. Front ends that can predict from one to three basic blocks were simulated, the number of instruction cache ports was varied from one to four. The systems with one and two ports predict one block per cycle; the ones with three ports predict two blocks, and the one with four ports predicts three blocks.

The averaged results are plotted in Figure 11 and the results for individual benchmarks are in Figure 9 speedups are shown in Figure 10. In all plots systems are arranged in order of increasing fetch rate The speedup shown is for a pre-execution system compared to a conventional one using the same front end, showing how much added performance can be obtained using pre-execution.

The reduction of pre-execution effectiveness with fetch rate, if present, is small. Several benchmarks show increased speedup with fetch rate as the impact

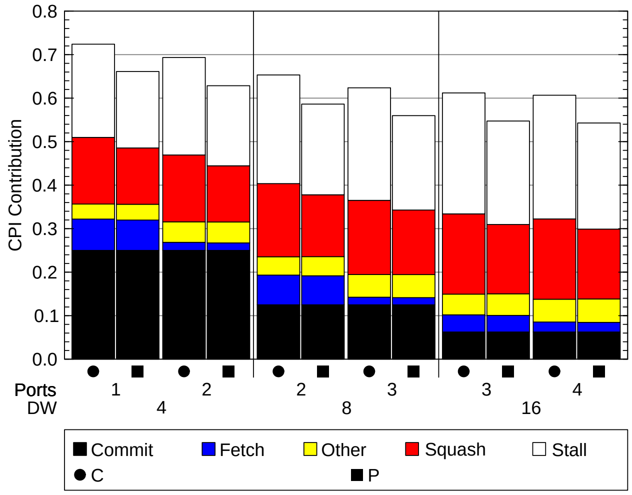

Figure 11. Average performance of conventional and pre-execution systems with different front ends. DW indicates decode width (from 4- to 16-way superscalar), Ports indicates number of instruction cache ports. Segments show CPI contribution.

of control- and window-critical loads becomes a larger fraction of execution time.

6. Related Work

Pre-execution schemes rely on a main thread to repeatedly trigger slices that only run briefly. In contrast the Slipstream Processor of Sundaramoorthy, Purser, and Rotenberg [17] and in systems using Master/Slave Speculative Parallelization of Zilles and Sohi [22] a condensed version of the original program serves the same purpose as a slice: it produces critical values faster than the original program would. Unlike a slice the condensed program is long running and produces more than just troublesome results. Its results are communicated to hardware running the original program where it is used for predictions and later checked for correctness. Correctness needs to be checked because the condensed program, in order to keep ahead of the original, may not always compute correct results.

Pre-execution is just one way of executing a load instruction early. Another is to allow fetch to proceed when some resource limit nears. Instructions fetched under these conditions may prefetch the cache and can provide branch outcomes. Balasubramonian, Dwarkadas, and Albonesi [2] describe such a scheme in which fetch goes on as a *future thread* even when the number of physical registers is low and can discard completed or unneeded instructions, avoiding the reorder buffer size limitation.

Pre-execution schemes are one way of starting loads early, hardware prefetch schemes are another [7]. In such schemes hardware monitors addresses that miss the cache, both level-1 and level-2 prefetchers have been investigated. The hardware, designed to recognize sequential [16], stride [5], or previously encountered patterns [6], generates a prefetch for a predicted next address in the pattern.

Hardware prefetch schemes are quite effective on many memory access patterns, but have trouble predicting many others. Several of the pre-execution schemes have been compared against hardware prefetch and found to complement it well, prefetching addresses that hardware prefetch could not [1,11].

7. Conclusions

Pre-execution improves performance by resolving load addresses and branch directions early using a thread that is forked speculatively upstream of the load. As verified here, pre-execution cannot be outrun by any reasonable front end, faster fetch results in only a small reduction in speedup.

Performance improvement is achieved by prefetching control- and window-critical loads. There are few alternative mechanisms to improve control-critical loads, but the impact of many window-critical loads can be reduced by increasing the reorder buffer size. Data presented here show that for some benchmarks, systems with larger reorder buffers enjoy the same speedup as those with pre-execution, the benchmarks so affected are the ones pre-execution is most effective on. When systems with larger reorder buffers also have longer scheduling delays pre-execution is at a greater advantage. The longer scheduling delays slow many programs while the larger ROB size helps only a few. This is in contrast to pre-execution which (as simulated) only slows one program. One factor not examined here is which is less costly, pre-execution or increased reorder buffer size.

8. Acknowledgments

This work was supported in part by the National Science Foundation under Award No. CCR-0105478 and through an allocation of time on the high-performance computing facilities within LSU's Center for Applied Information Technology and Learning, which is funded through Louisiana legislative appropriations.

9. References

[1] Murali Annavaram, Jignesh M. Patel, and Edwared S. Davidson, "Data prefetching by dependence graph pre-computation," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, June 2001, pp. 52-61.

[2] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H. Albonesi, "Dynamically allocating processor resources between nearby and distant ILP," in *Proceedings of the 28th Annual International Symposium on Computer Architecture,* June 2001, pp. 26-37.

[3] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen, "Dynamic speculative precomputation," in *International Symposium on Microarchitecture,* December 2001, pp. 306-317.

[4] A. N. Eden and T. Mudge, "The YAGS branch prediction scheme," International Symposium on Microarchitecture, December 1998, pp. 69-77.

[5] J. Fu, J.H. Patel, and B.L. Janssens, "Stride directed prefetching in scalar processors," in *Proc. of the 25th Annual International Symposium on Microarchitecture*, pp. 102-110, 1992

[6] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *Proc. of the Int'l Symp. on Computer Arch,* June 1997, pp. 252–263.

[7] Steven P. Vanderwiel and David Lilja, "Data prefetch mechanisms," *ACM Computing Surveys*, vol. 32, no. 2, pp. 174-199, June 2000.

[8] Jack L. Lo, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, and Dean M. Tullsen, "Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading," *ACM Transactions on Computer Systems,* vol. 15, no. 3, pp. 322-354, August 1997.

[9] Chi-Keung Luk and Todd C. Mowry, "Compiler-based prefetching for recursive data structures," in the proceedings of *The Seventh International Conference on Architectural Support for Programming Languages and Operating Systems,* October 1996, pp. 222-233.

[10] Chi-Keung Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," in *Proceedings of the 28th Annual International Symposium on Computer Architecture,* June 2001, pp. 40-51.

[11] Andreas Moshovos, Dionisios N. Pnevmatikatos, and Amirali Baniasadi, "Slice-processors: an implementation of operation-based predictoin," in *Proceedings of the 15th International Conference on Supercomputing,* 2001, pp.321-334.

[12] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve, "RSIM reference manual version 1.0," Rice University Dept. of Electrical and Computer Engineering, August 1997, Technical Report 9705.

[13] Eric Rotenberg, Steve Bennett, and James E. Smith, "A trace cache microarchitecture and evaluation," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 111-120, February 1999.

[14] Amir Roth and Gurindar Sohi, "Speculative data-driven multithreading," in *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, 2001.

[15] Amir Roth and Gurindar Sohi, "A quantitative framework for automated pre-execution thread selection," in *Proc. of the 35th Annual International Symposium on Microarchitecture*, pp. 430-441, 2002

[16] A.J. Smith, "Sequential program prefetching in memory hierarchies," *IEEE Computer*, vol. 11, no. 12, pp.7-21, Dec. 1978

[17] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg, "Slipstream processors: improving both performance and fault tolerance," in the proceedings of *The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems,* November 2000, pp. 257-267.

[18] David L. Weaver and Tom Germond (eds.), "The SPARC architecture manual, Version 9," Englewood Cliffs, New Jersey: Prentice-Hall, 1994.

[19] Tse-Yu Yeh, Deborah T. Marr, and Yale N. Patt, "Increasing the instruction fetch rate via multiple branch prediction and a branch address cache," in *Proceedings of the International Conference on Supercomputing,* 1993, pp. 67–76.

[20] Craig Zilles and Gurindar Sohi, "Understanding the backward slices of performance degrading instructions," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000, pp. 172–181.

[21] Craig Zilles and Gurindar Sohi, "Execution-based prediction using speculative slices," in *Proceedings of the International Symposium on Computer Architecture,*, June 2001, pp. 2-13.

[22] Craig Zilles and Gurindar Sohi, "Master/slave speculative parallelization," in *Proc. of the 35th Annual International Symposium on Microarchitecture*, pp. 85-96, 2002