Accurately Approximating Superscalar Processor Performance from Traces

Kiyeon Lee, Shayne Evans, and Sangyeun Cho
Dept. of Computer Science
University of Pittsburgh
{lee,sevans,cho}@cs.pitt.edu

Abstract

Trace-driven simulation of superscalar processors is particularly complicated. The dynamic nature of superscalar processors combined with the static nature of traces can lead to large inaccuracies in the results, especially when traces contain only a subset of executed instructions for trace reduction. The main problem in the filtered trace simulation is that the trace does not contain enough information with which one can predict the actual penalty of a cache miss. In this paper, we discuss and evaluate three strategies to quantify the impact of a long latency memory access in a superscalar processor when traces have only L1 cache misses. The strategies are based on models about how a cache miss is treated with respect to other cache misses: (1) isolated cache miss model, (2) independent cache miss model, and (3) pairwise dependent cache miss model. Our experimental results demonstrate that the pairwise dependent cache miss model produces reasonably accurate results (4.8% RMS error) under perfect branch prediction. Our work forms a basis for fast, accurate, and configurable multicore processor simulation using a pre-determined processor core design.

1. Introduction

Simulation is an important tool for computer architects [26]. It enables one to quickly analyze the behavior of a complex system and to evaluate subtle design trade-offs in an experimental environment. However, its use is limited to situations where it is both reasonably accurate and fast. *Trace-driven simulation* is a particularly fast simulation method consisting of two phases [22, 26]. In the *trace generation* phase a benchmark is executed and information about key events is recorded in a trace file. In the *trace simulation* phase, the information recorded in the first phase is used to drive the simulation. Trace-driven simulation's increased speed is a result of replacing the detailed functional execution of a benchmark with a pre-captured, but highly representative, trace of an execution.

This technique works well for in-order single-issue cores. For example, consider the detailed simulation of a simple in-order core. During the trace generation phase, one might

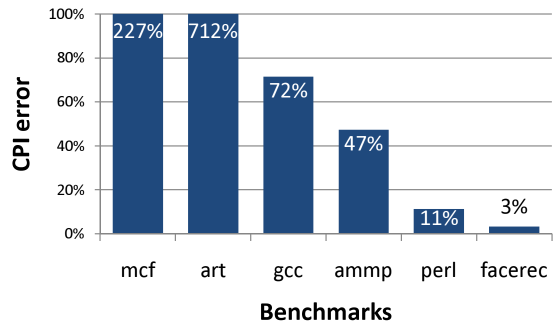


Figure 1. Naïve trace-driven simulation of a superscalar processor may suffer from very poor accuracy.

record the type and address of every memory operation as well as the number of instructions executed and the number of cycles elapsed since the last memory operation. This *filtered trace* only includes memory accesses, a subset of the instructions, and summarizes the instructions executed between operations. Because the core executes instructions in order and blocks while waiting for a memory access, the filtered trace would be the same regardless of the memory configuration. Thus using the same trace, one could simulate many different memory hierarchy configurations, such as different cache latencies or cache sizes, with high cycle accuracy and fast speed.

However, many issues arise when the same scheme is applied to out-of-order superscalar processors. For example, a superscalar processor does not necessarily block during a memory access. Modern processors often execute instructions during the memory access latency to hide the cost of the latency. A complex processor core dynamically chooses which instructions to execute, but a trace naturally contains the choices made by a core in one particular instance of execution. When parameters are varied outside of the core, the trace loses its accuracy. This makes naïve trace-driven simulation of superscalar processor difficult. A naïve trace-driven simulation sequentially processes each trace item in the order in the trace file. It adds the elapsed cycles recorded in the trace items and the latencies returned from accessing the L2 cache or memory to the total simulated clock cycle. Figure 1, produced using a typical 4-issue processor model, shows that such an approach results in very high errors.

Accurate trace-driven simulation of superscalar proces-

[1]Shayne Evans is currently with the Lime Brokerage LLC.

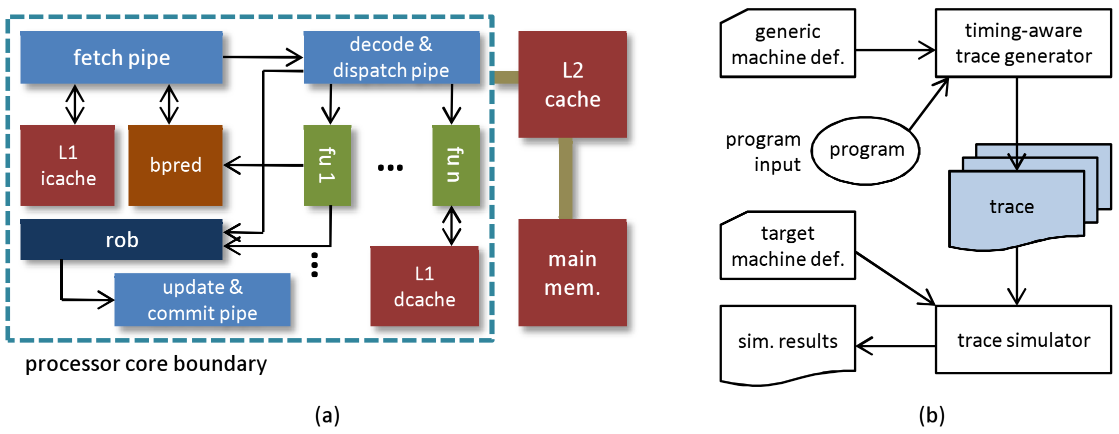


Figure 2. (a) Machine model having a superscalar processor core, L2 cache, and main memory. (b) Trace-driven simulation setup.

sors is possible using *full traces* [3]. A full trace includes a detailed trace item for every instruction. This enables the processor model to determine dependencies between instructions and mimic the choices of an actual superscalar processor without executing a benchmark. This paper, advocates *accurate trace-driven simulation using filtered traces* because it simplifies the complexity of modeling a processor core, obtains results faster, and requires less storage space than traditional trace-driven simulation using full traces. The main problem is: When simulating a superscalar core using a filtered trace it is difficult to obtain accurate results because the trace does not contain enough information to faithfully predict the core's dynamic behavior.

This paper argues that with an understanding of the characteristics of superscalar processors, one can produce both fast and reasonably accurate simulation results using filtered traces. Specifically, we present and evaluate three strategies to quantify the impact of a long latency memory access in a superscalar processor when traces have only L1 cache misses (i.e., L1 cache hits are filtered). They are based on three different models about how we treat a cache miss with respect to other cache misses: (1) isolated cache miss model, (2) independent cache miss model, and (3) pairwise dependent cache miss model.

Our results show that the pairwise dependent cache miss model obtains sufficiently small simulation errors (4.8% RMS) at very high simulation speeds (over 30.7× on average) compared with the detailed execution-driven simulation approach under perfect branch prediction. Given the fast simulation speeds and small simulation errors, our approach is particularly useful for studying large-scale multicore processors at an early design stage when the focus in a series of simulation studies is on system-wide parameters such as the on-chip interconnection network and L2 cache configurations rather than a superscalar processor's internal parameters such as the L1 cache configuration, issue width, and reorder buffer size. We expect that recent trace-driven multicore processor simulators such as Zauber [15] and TPTS [7] can readily integrate our techniques.

The rest of this paper is organized as follows. Section 2 describes our superscalar machine model and experimental setup. The three proposed cache miss models are presented and evaluated in Sections 3, 4, and 5, respectively. In Section 6 we compare the proposed models to place them in perspective and discuss how our techniques can be applied to fast multicore processor simulation. Section 7 summarizes related work and in Section 8 we provide our conclusions.

2. Machine Model and Experimental Setup

2.1. Machine model

Our machine model is a superscalar processor system with two levels of cache memory and a main memory, as shown in Figure 2(a). The superscalar processor core model we use is sketched inside the dotted box. It has a front-end "fetch pipeline" that fetches and buffers instructions for further processing. Once fetched, instructions are decoded and dispatched to various functional units such as an ALU, branch unit, or data memory access unit. They may be temporarily stored in buffers (or *reservation stations*) associated with a specific functional unit until the unit becomes available or input operands arrive. When an instruction is dispatched, a new entry in the reorder buffer (ROB) is allocated so that the "update and commit pipe" can change the architectural state properly in the program order as instructions are committed in the presence of special events such as exceptions, branch mispredictions, and cache misses. More general description of superscalar processor design and operation can be found in Johnson [9] and Shen and Lipasti [19].

When a memory access misses in an L1 cache inside the processor, it accesses an off-core unified L2 cache. If the access misses in the L2 cache, it will access the main memory. It is noted that the L2 cache and main memory are considered a system-wide resources in multicore processor architectures [2, 14] because they are often shared among processor cores whereas L1 caches are not.

2.2. Experimental setup

As a trace-driven simulation framework, our experimental setup employs two distinct tools, a *trace generator* and a *trace simulator*. Unlike most previous trace-driven simulation work [22], we introduce the notion of *timing* during the trace generation phase and embed time-related information in the trace. Similar timing-aware trace generation was done in recent trace-driven multicore simulators [7, 15]. Hence, the trace generator must be able to model the microarchitecture of a processor using a user provided machine definition. Figure 2(b) illustrates the relationship between the trace generator, trace simulator, machine definitions, and trace files. The *generic machine definition* refers to the superscalar processor core configuration that we use: the intra-core parameters that shape the processor core in Figure 2(a). The *target machine definition* is the system-level processor configuration such as L2 cache size, associativity, and main memory latency, that completes the overall machine model we wish to study. Throughout this paper, we use `sim-outorder`, a detailed out-of-order processor simulator (of the SimpleScalar tool set) [1] to generate traces.

We use *filtered traces* in this work. That is, trace files do not contain all instructions executed during a program run and rather focus on memory access instructions [22]. Moreover, we filter out L1 cache hits that do not access the L2 cache, further cutting down the number of trace items to store in trace files, similar to [5,7,15]. Our traces are *timing-aware*; each trace item carries information about when it can be processed with regard to the preceding trace item. More specifically, each trace item captures: (1) how many instructions were executed after the last trace item, (2) how many cycles were spent executing those instructions, (3) information about the L1 cache miss that gave birth to the trace item: type (read, write, or instruction fetch) and its address, and finally (4) timing-related information to be used when a long latency memory access is caused by the trace item. This timing-related information depends on the generic machine model used in the trace generation phase.

Table 1 captures our baseline machine configuration. We note that we will use a perfect branch predictor in this paper to isolate the interference caused by branch mispredictions and avoid any confusion thereof. In this paper, we will only focus on quantifying the impact of long-latency memory access caused by a cache miss.

For experiments in the following three sections we use a selected set of SPEC2k benchmarks [21] for clear presentation: *mcf*, *art* (benchmarks with high miss rates), *gcc*, *ammp* (with medium miss rates), *perl* and *facerec* (with low miss rates). Selection was based on their L1 cache miss rates and the raw instruction level parallelism (ILP) present in the programs, such that strengths and weaknesses of the studied strategies can be exposed. In our current implementation, we treat "delayed hits" (hits to a cache block that is still in transit from the L2 cache) as trace items. We will present results for all SPEC2k benchmarks in Section 6. Programs were compiled using the Compaq Alpha C compiler (V5.9) with the `-O3` optimization flag. For each simulation, we skip the initialization phase of the target program [20], warm up caches in the next 100M instructions, and simulate the next 1B instructions. To evaluate the studied simulation methods, we use *CPI error* as the main metric. The CPI error is defined as $(T_{tsim} - T_{esim})/T_{esim}$, where T_{tsim} and T_{esim} are the simulated program execution time (number of cycles) of trace-driven simulation and execution-driven simulation respectively.

Dispatch/issue/commit width	4
Reorder buffer (ROB)	64 entries
Integer ALUs	4
Floating point ALUs	2
L1 i- & d-cache	1 cycle, 16KB, 4-way 64B line size, LRU
L2 cache (unified)	12 cycles, 1MB, 8-way 64B line size, LRU
Branch prediction	Perfect
Main memory latency	300 cycles

Table 1. Baseline machine configuration for experiments.

3. Model 1: Isolated Cache Miss

3.1. Basic idea

The basic idea of this model is quite simple: The actual impact of a particular cache miss on the overall program execution time is the time difference of two program runs, one without the miss and one with the miss, assuming that all other memory access latencies are unchanged.

Figure 3(a) captures this idea. Program run 1 has no L2 cache misses, whereas program run 2 of the same program has a single L2 cache miss at a known L2 cache access. The impact of the cache miss on the program execution time is simply $(T_{\text{run }2} - T_{\text{run }1})$. We can obtain $T_{\text{run }1}$ using a cycle-accurate simulator modeling an L2 cache having a 100% hit rate. $T_{\text{run }2}$ can be obtained by using the same cycle-accurate simulator and giving the miss penalty to a specific L2 cache access. One can measure the impact of each potential L2 cache miss by repeating this process.

3.2. Instruction permeability analysis

While the basic idea of our isolated cache miss model is intuitive, the process of assessing the impact of each potential L2 cache miss can be extremely time consuming. Suppose that a program has N L1 cache misses. In an exhaustive approach to analyze this program, for instance, one will generate N traces (each having exactly one L2 cache miss) and compare them against the trace having no L2 cache misses to deduce the impact of each individual cache miss.

To reduce the overhead of generating many traces to com-

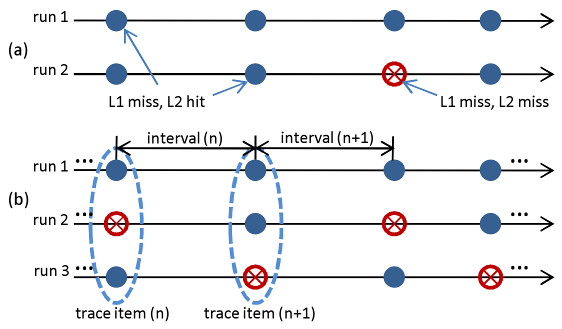


Figure 3. (a) A single "isolated" L2 cache miss in a program run. (b) Using two additional traces generated by interleaving hit and miss to compute the impact of isolated misses efficiently.

pute the impact of each potential L2 cache miss, we use a technique called *instruction permeability analysis* that systematically assigns a cache access latency to trace items as they are generated in the trace generation phase. Figure 3(b) shows three traces generated from a target program for instruction permeability analysis. One trace has only L2 cache hits and the other two have alternating L2 cache hits and misses. The alternation of cache hits and misses is skewed in the two traces so that all trace items are covered. By comparing the actual number of cycles measured in trace intervals, each surrounded by two trace items, we can compute the impact of a single L2 cache miss as we would do with a trace having only a single L2 cache miss. We call the configuration in Figure 3(b) 2-interleaving because the additional traces have one L2 cache miss on every two trace items.

Let us turn our attention to how we actually analyze the impact of a cache miss and how we associate the information with trace items. Assume that S is the latency of a cache hit and L is the latency of a cache miss. L is the latency penalty paid on a specific cache miss (i.e., main memory access) on top of a cache access latency S. From measurements one can obtain a, the cycle count of the interval (n) after trace item (n) in trace 1 and b, the cycle count of the same interval in trace 2. We define $d_n = b - a$. Because the nth trace item in trace 2 has a longer latency $(S + L)$ than the corresponding trace item in trace 1 (S), $b \geq a$ holds and equivalently $d_n \geq 0$. Once we obtain d_n, we annotate trace item (n) with the timing information (a, Δ_n) where Δ_n is defined as $(L - d_n)$. Given this, the actual latency of interval (n) during the trace-driven simulation is:

$$\begin{array}{ll} a & \text{if trace item } n \text{ hits in L2 cache and} \\ a + L' - \Delta_n & \text{if trace item } n \text{ misses in L2 cache} \end{array}$$

where L' is the actual main memory access latency used in the trace-driven simulation. When $L = L'$, our method guarantees that the actual latency computed for interval (n) is a or b depending on the cache access outcome of trace item (n), the same as those of the timing-aware trace generation. If $L \neq L'$, the actual latency will be either a or $(b+(L'-L))$.

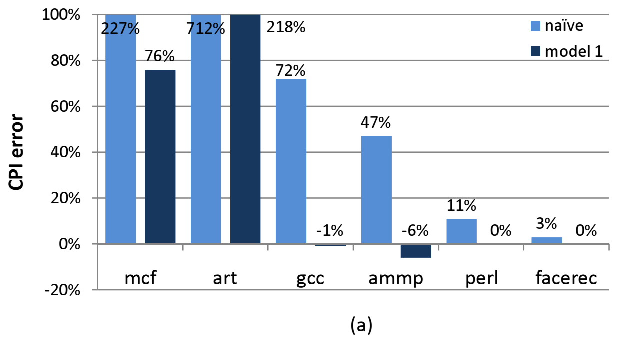
The above description of instruction permeability analysis used a 2-interleaving configuration. One may choose to employ a 3-interleaving configuration where there is one L2 cache miss on every three trace items. Obviously, the most important factor affecting the effectiveness of this scheme is how far in time trace items are separated from each other. If a "missed" trace item is far away from the next missed trace item in trace 2 and 3 in the example of Figure 3(b), the result of the analysis will be a close approximation of what we would get from the exhaustive method. Hence, we expect that an n-interleaving configuration will result in higher accuracy than an m-interleaving configuration if $n > m$, at a higher trace generation and analysis cost. If $n = N$ where N is the number of trace items, the n-interleaving configuration degenerates to the exhaustive method.

3.3. Result

During experiments, we found that it is rather challenging to correctly align matching trace items from multiple trace files to perform instruction permeability analysis, especially at a high interleaving factor. This is because the order of trace items is not preserved across the trace files as we assign different cache access latencies to different trace items. Certain trace items occur in one trace file, but not in others, mis-aligning the trace items that follow. While devising better trace item annotation methods is certainly an interesting question, we limit our presentation in this section to the 2-interleaving configuration, improved with an ad hoc method that uses a few more traces. With four more trace files where long-latency trace items were chosen randomly, we could annotate 56% (*mcf*), 66% (*art*), 75% (*gcc*), 73% (*ammp*), 98% (*perl*), and 100% (*facerec*) of the trace items.

Despite being able to considerably reduce the magnitude of CPI errors compared with the naïve method, Figure 4(a) shows that the isolated cache miss model fails to eliminate errors robustly. Programs having a high L1 cache miss rate (*mcf* and *art*) still see a large CPI error. These programs have many independent, parallel cache misses in short intervals, resulting in incorrect accumulation of cache miss penalties.

Figure 4(b) shows that the studied programs have many L2 cache accesses in short intervals, confirming our observation. *Facerec* has a low L1 data cache miss rate and its L1 cache misses occur sparsely. This makes the isolated cache miss model (and even the naïve method) work well for *facerec*. Interestingly, *gcc* and *ammp* have a negative CPI error, which was caused by our aggressive ad hoc trace item annotation. We employed a search-based trace item matching algorithm that exhaustively inspects trace items within a specified range until it finds the matching interval given two trace items. Some annotations (Δ), especially in trace items that exhibit different ordering in different trace files, become inaccurate and often larger. As a result, at simulation time, the computed penalty for cache misses that occur from the corresponding trace items becomes smaller.



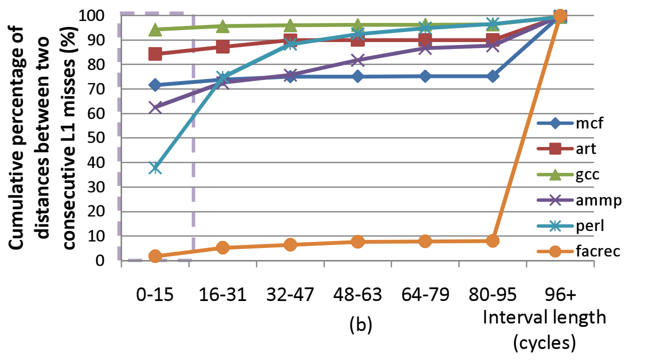


Figure 4. (a) CPI error of the isolated cache miss model. (b) The cumulative percentage of L1 miss intervals in terms of clock cycles. All L1 misses are assumed to hit in the L2 cache. The dotted box shows that there are potentially many independent L2 cache accesses. The X-axis shows the number of cycles in each L1 miss interval.

4. Model 2: Independent Cache Miss

4.1. Basic idea

The main weakness of the isolated cache miss model lies in its assumption that the impact of a long latency memory access is *accumulated*. Hence, while the model is capable of accurately quantifying the delay penalty of a relatively "isolated" cache miss, it loses accuracy when cache misses are close to each other; it pessimistically adds individually computed delay penalties even if those misses can be overlapped ("memory level parallelism") in a real processor.

Unlike the isolated cache miss model, the *independent cache miss model* is optimistic about when an L1 cache miss in a trace item can proceed to the L2 cache. It assumes that all L1 cache misses are *independent* of each other and can be handled without regard to any outstanding cache misses. The independent cache miss model can potentially result in more accurate results than the isolated cache miss model because it enables a trace-driven simulator to process multiple cache miss events simultaneously (rather than sequentially) as a superscalar processor would do.

It is evident that there is a limit on how many L1 cache misses can be pending at any given time considering a superscalar processor's limited hardware data structures. For example, our processor configuration in Table 1 has a 64-entry ROB and hence will not allow two memory instructions to be simultaneously outstanding if they are at least 64 instructions away from each other. Hence, under the independent cache miss model we take each L1 cache miss independently and make it proceed only when such a decision does not contradict with the processor state that has been constructed up to the point of the cache miss. In this paper, we focus on the ROB among many processor data structures based on our own experiments and a previous analytical performance modeling work done by Karkhanis and Smith [11].

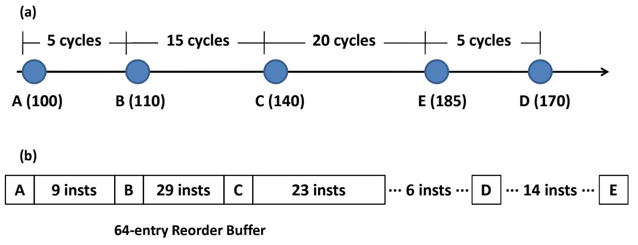
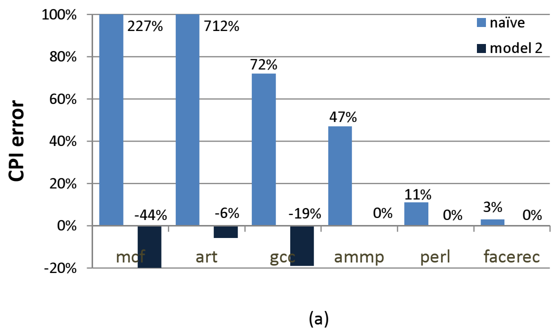


Figure 5. (a) Five trace items recorded in the trace file with information collected during the trace generation phase. Inside parentheses are the instruction sequence numbers. (b) The status of the ROB: Only the first three trace items are in the ROB.

4.2. ROB occupancy analysis

To implement the idea of the independent cache miss model, we need to model the ROB in our trace-driven simulation. In what follows, we will illustrate how we determine the progress of each trace item using the proposed analysis technique via an elaborate example.

Figure 5 depicts an example of how the instruction sequence number attached to each trace item is used to determine when to process the trace items. In the example, all five trace items (i.e., L2 cache accesses) miss in the L2 cache and go to the main memory. A is the oldest (memory) instruction inside the ROB waiting for its data to come from the memory. B and C are subsequent memory instructions that are issued while A is pending. However, since the number of instructions between C and D is larger than the available free entries in the ROB, some instructions in D cannot be issued until there is room in the ROB to hold them. After A commits, the issued instructions between A and B are committed, which allows the instructions between the tail of the ROB and D, as well as the instruction in D, to advance to the ROB. However, the available entries are still not enough to hold all 14 instructions between D and E, and only the next three instructions behind D are placed in the ROB. After B is resolved and commits, the instructions between B and C will



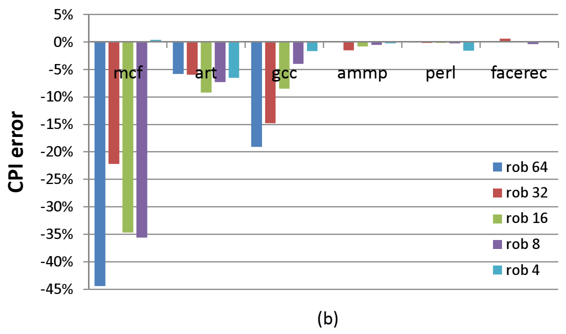


Figure 6. (a) CPI error of the independent cache miss model when ROB size is 64. (b) CPI error with different ROB sizes.

follow and commit at the processor's commit rate. As more and more entries become free, E will finally move into the ROB and be issued. In essence, the ROB occupancy analysis keeps track of instructions in the ROB after each successive trace items, allows all L2 cache accesses in the ROB to issue independently, and blocks any further processing of the following trace items if the ROB is full.

The ROB occupancy analysis considers the instruction sequence number ($inst_{seq}$) and the number of elapsed cycles ($cyc_{elapsed}$) between two trace items to determine when to process the trace items. According to the ROB occupancy analysis, L2 cache misses are treated in the following way. If $inst_{seq}$ of the next trace item is smaller than the maximum $inst_{seq}$ in the ROB, this indicates that the next trace item is already inside the ROB, and hence, the instruction in the next trace item can be issued independently. If $inst_{seq}$ of the next trace item is larger than the maximum $inst_{seq}$ in the ROB and the difference between the two is smaller or equal to the number of free entries, the instructions in the trace item are moved into the ROB at a speed determined by $cyc_{elapsed}$. The number of instructions inside the ROB can be calculated by comparing the head and tail of the ROB. When a trace item cannot be immediately processed as in the case of D and E of the above example, it has to wait until presently outstanding L2 misses are resolved.

The ROB occupancy analysis is done within the trace-driven simulation. Therefore, the independent cache miss model does not require any trace analysis before simulation. Note that the isolated cache miss model requires that multiple traces be generated and analyzed.

4.3. Result

Figure 6(a) compares the result of the independent cache miss model and `sim-outorder`. It shows that the program execution times obtained by the independent cache miss model are in general smaller than those of `sim-outorder` (negative CPI errors). This is because the independent cache miss model is optimistic about when a trace item can be processed (i.e., L2 cache is accessed) and aggressively processes memory accesses in parallel. In the case of *mcf* we observed a large simulated execution time deviation. We attribute this large magnitude of error to the memory access pattern of *mcf*–there are many trace items that are dependent on other trace items. Our profiling reveals that 80% of *mcf*'s trace items have dependencies which are neglected in the independent cache miss model.

We saw the largest improvement in *art* compared to the naïve method because the majority of its L1 cache misses occur very closely to each other, as indicated in Figure 4(b). This suggests that there are potentially many independent L2 cache misses in *art* which are accurately quantified using the independent cache miss model.

Finally, we note that the degree of MLP is constrained by the ROB size, thus affects the accuracy of the independent cache miss model. Figure 6(b) depicts how CPI errors change when the ROB size is varied. The CPI error tends to decrease as we reduce the ROB size. In *mcf*, the large CPI error persisted until the ROB size was reduced to 4. Besides *art* showing -7% CPI error, other five benchmarks showed a CPI error that is smaller than -2% (0% for *mcf*) when the ROB size is four, since memory accesses that have interdependence are often not placed in the ROB together (and are not issued together).

5. Model 3: Pairwise Dependent Cache Miss

5.1. Basic idea

The independent cache miss model we studied in the previous section can be too optimistic. It works well for the programs that have few dependencies between cache misses but it results in smaller program execution times by scheduling memory accesses aggressively. On the other hand, the isolated cache miss model in Section 3 is pessimistic about the dependences between trace items and processes them sequentially. Hence, the impact of each long-latency memory access is simply accumulated. This approach works well for the programs that inherently have few parallel memory accesses but it results in large CPI errors for the programs that have many independent memory accesses that are clustered.

In this section, we propose yet another model that combines the strengths of the two previous models. The new model exploits the parallel scheduling capability of the in-

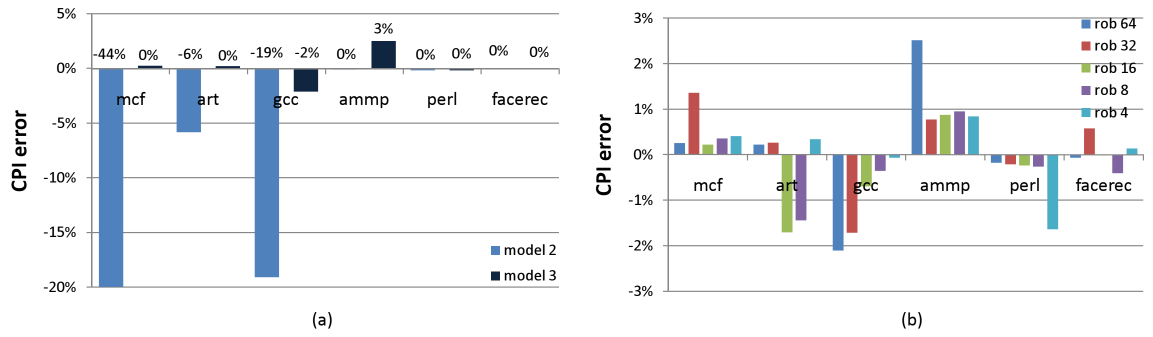


Figure 7. (a) CPI error of the independent cache miss model and the pairwise dependent cache miss model when the ROB size is 64. (b) CPI error of the pairwise dependent cache miss model with different ROB sizes.

dependent cache miss model as well as the explicit dependencies between trace items collected during the trace generation phase. Unlike the independent cache miss model that was shown to be "overly optimistic" for some benchmarks, the pairwise dependent cache miss model honors dependencies between trace items when scheduling them. To do so, in the trace generation phase we detect dependencies between trace items and record them. In the trace simulation phase, if a trace item in the ROB depends on a previous trace item (i.e., ancestor), it is not issued until the previous ancestor trace item gets its data back from the cache or memory. Hence, if there exists a dependency between two trace items, even if both of them are already in the ROB, the second, the dependent cache access is not processed immediately.

To detect dependencies among trace items efficiently, we can exploit the existing dependence analysis facilities in a superscalar processor simulator. In the modified `sim-outorder` simulator we use during trace generation, we take advantage of the "dependency chains" constructed when instructions are dispatched. Specifically, to determine if there is a dependency between a pair of distinct memory accesses, we walk through the dependency chains to see if there exists a linkage between them. Detected dependencies are then recorded in trace items. We note that a single trace item may depend on multiple preceding trace items. However, our experiments show that storing more than a single ancestor does not produce significantly better results and thus, we store only one ancestor or none (no dependence) in each trace item. In the presence of multiple ancestors, we use a heuristic to choose the latest ancestor in the instruction sequence, the closest to the trace item under consideration.

5.2. Result

Figure 7(a) compares the accuracy of the independent cache miss model and the pairwise dependent cache miss model. It is shown that the pairwise dependent cache miss model significantly reduces the errors of the independent cache miss model. All the programs we examined have a CPI error that is within $\pm 3\%$. When the independent cache miss model was used, *mcf* had a large CPI error (-44%). The proposed pairwise cache miss model, by exploiting the dependence information embedded in trace items, reduced the CPI error from -44% to 0% for *mcf* . *Facerec* and *perl* have few dependencies between trace items, hence the CPI error did not change.

Figure 7(b) shows how the CPI error of the pairwise dependent cache miss model changes when the ROB size is varied. As was the case with the independent cache miss model, the CPI error becomes smaller as we reduce the ROB size. When the ROB size is as small as four, the processor starts to behave like an in-order processor and both the independent cache miss model and the pairwise dependent cache miss model give very small CPI errors compared with the execution-driven simulation.

6. Putting It All Together

6.1. Comparing three cache miss models

The three cache miss models we have examined so far have different strengths and weaknesses. The isolated cache miss model works well when the simulated program has a high L1 cache hit rate and L1 cache misses are "isolated" and occur apart from each other. It is pessimistic about how trace items (cache misses) can be scheduled during simulation; a long latency cache miss will simply block and delay all following trace items. It also requires that the potential penalty of individual cache misses be pre-calculated before simulation during the trace generation phase. The related analysis entails generating multiple traces and comparing trace items in those traces. The process was shown to be error-prone for programs that have many clustered misses.

The independent cache miss model is optimistic about when a trace item can be scheduled; trace items are processed immediately as long as there is space in the ROB to hold them. It produces much smaller CPI errors than the isolated cache miss model when cache misses occur frequently and the misses overlap in time in a real superscalar processor. This model does not require any pre-analysis of traces. Traces simply capture the L1 cache misses and the trace sim-

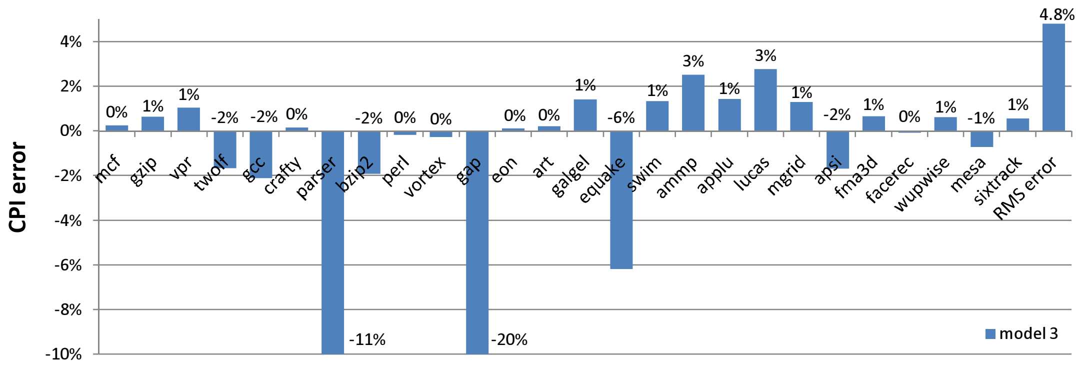


Figure 8. CPI error of all 26 SPEC2k benchmarks using the pairwise dependent cache miss model and the baseline machine configuration.

ulator would determine the timing of each trace item in the trace using the ROB status constructed on the fly. However, if there are dependencies among trace items, this overly optimistic model becomes inaccurate.

Our third model, the pairwise dependent cache miss model, builds on the independent cache miss model. It considers dependencies between trace items as it schedules them. If a benchmark program does not present dependencies between trace items, the pairwise dependent cache miss model behaves just like the independent cache miss model. In other cases, it reduces the CPI error of the independent cache miss model by properly delaying trace items that depend on an unresolved trace item. Figure 7(a) showed that the pairwise dependent cache miss model consistently outperforms the independent cache miss model in terms of the CPI error metric. We conclude that the pairwise dependent cache miss model is the most accurate model among the three when modeling the superscalar processor performance from traces. Figure 8 presents the CPI error of all SPEC2k benchmarks, produced using the pairwise dependent cache miss model when the ROB size is 64. As many as 23 out of 26 programs had an error that is less than 5%. The average CPI error was 4.8% RMS.

Our models are fed with filtered traces that contain only L2 cache accesses. This filtering method is simple and effective in reducing the trace file size. The average number of instructions in trace items (# of instructions before hitting an L1 miss) ranged from 5 (*gcc*) to 1,418 (*sixtrack*). The maximum number of instructions in a trace item was between 162 (*art*) to 65,513 (*perl*), and the minimum number of instructions was 1 for all benchmarks. We note that the average number of instructions represented in a trace item will increase if we make the L1 data cache larger in the simulated machine configuration. This will in turn reduce the trace file size. In our study, the L1 cache size was 16KB.

Trace-driven simulation using filtered traces is fast. We have observed simulation speedups roughly comparable to those reported in [7, 15]. The observed simulation speedups range from 2 (*gcc*) to 306 (*gap*) and their average (geometric mean) was 30.7.

6.2. Toward efficient multicore system simulation

We have previously used "CPI error" as the metric to evaluate how closely our trace-driven simulation approach approximates a superscalar processor's performance compared with an execution-driven simulator, given the same benchmark program and the same machine configuration. In this subsection, we will evaluate two other aspects of our trace-driven simulation model: (1) When a processor core is modeled using our approach, does it change how the core exercises system-wide resources such as shared cache and on-chip network? and (2) Can our scheme predict a program's relative performance when a system parameter (such as L2 cache size) is changed? These aspects are especially important for accurate and efficient multicore simulation. The efficiency of our approach is its simulation speed advantage over the execution-driven simulation approach.

To examine the first aspect, Figure 9 compares the L2 cache miss intervals of the two simulators, `sim-outorder` and our trace-driven simulator using the pairwise dependent cache miss model. Our intuition here is that they should produce similar histograms if our model faithfully simulates the superscalar processor from traces. Overall, the pairwise dependent cache miss model preserves the memory system access behavior of `sim-outorder` closely. In *mcf*, some very short intervals ("0–15") have shifted into the next, longer interval range ("16–31"). In *gcc*, some of the very long intervals ("208+") have shifted into relatively short intervals. Using the percentage of common populations across different bins as the metric, the similarity of the two simulator is: 95.8% (*mcf*), 93.5% (*gcc*), 99.7% (*perl*), 94.6% (*art*), 98% (*ammp*), and 100% (*facerec*).

We now attempt to answer the second question "Can the proposed pairwise dependent cache miss model correctly predict the performance of a new machine configuration given the performance of a baseline machine configuration?" The ability to measure relative performance is important in a multicore system performance study where some resources

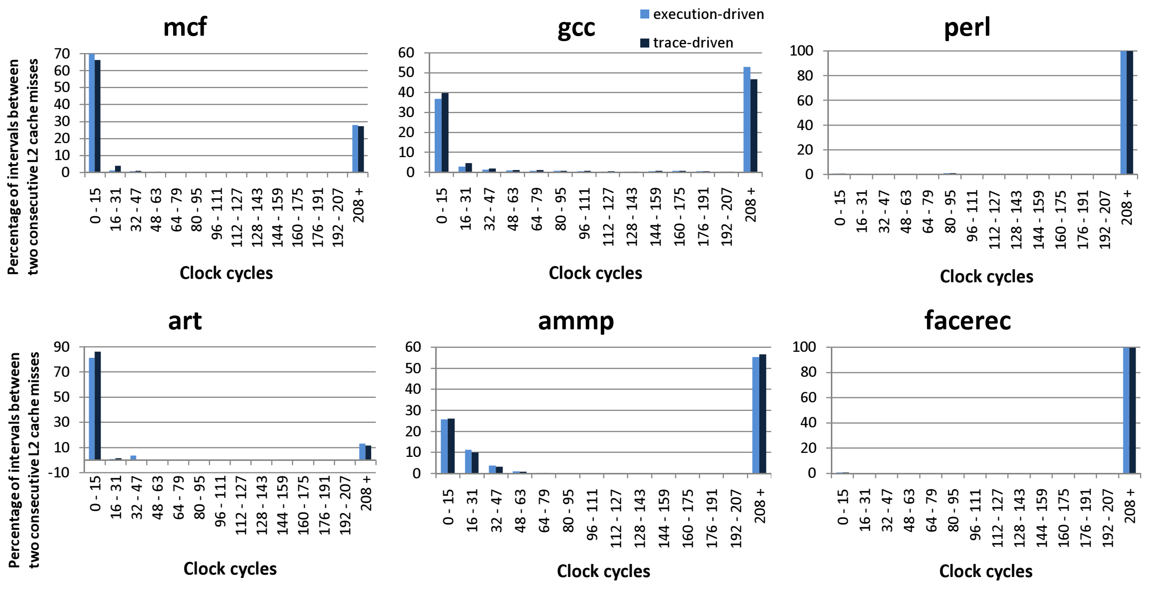


Figure 9. Comparing the distance (in clock cycles) between two consecutive L2 cache misses returned from the detailed execution-driven simulation (`sim-outorder`) and our pairwise dependent cache miss model.

are shared among threads. For instance, Kim et al. [12] uses "relative slowdown," the ratio of a program's execution time on a "shared" multicore system to the same program's execution time on an "unshared" system. We make our baseline configuration in Table 1 as the reference point and simulate five configurations that differ in one of their L2 cache or main memory parameters (described in Table 2). Note that our trace-driven simulations used the same traces produced to study the baseline configuration. On the other hand, we ran `sim-outorder` with each new machine configuration examined. The numbers in Table 2 are relative execution time differences of the two simulators. Relative performance is defined as the ratio of the execution time on a given configuration to the execution time on the baseline configuration.

Our results show that the trace-driven simulator was able to project the relative performance very closely to the execution-driven simulator, `sim-outorder`. First of all, the performance direction (positive or negative), was predicted correctly 100% of the time. Furthermore, Table 2 shows that the magnitude of relative performance seen by each benchmark and each configuration, was nearly identical between the two simulators. The average RMS errors were less than or equal to 2.0% across all the configurations examined.

In summary, the results presented in Figure 9 and Table 2 suggest that the trace-driven simulation method based on the pairwise dependent cache miss model is amenable for use in a trace-driven multicore simulation environment [7, 15]. To simulate multiple processor cores that run independent threads simultaneously (i.e., multiprogrammed workload), one can prepare traces from a detailed uniprocessor simulator (like `sim-outorder`) and run them together. Our techniques can be applied to multithreaded shared memory applications if individual threads can be traced [7]. One can reliably study the overall system behavior thanks to the capability of our techniques to preserve each processor core's memory access behavior like an execution-driven simulation engine. At the same time, one can examine how individual program performance is affected by contentions in the shared resources.

7. Related Work

Trace-driven simulation has been an indispensable technique for analyzing computer performance [22, 26]. Our work is a positive response to the question "Can trace-driven simulators accurately predict superscalar performance?" posed by Black et al. [4]. In their work in 1996, they determined that sampling techniques present a problem to the accuracy of trace-driven simulation for superscalar processors, and questioned the accuracy of trace-driven simulation for superscalar processors even when the full traces were used as the processor complexity continues to increase and the benchmarks evolve to run for longer times.

In previous and current practice, much trace-driven simulation work has focused on either tracing memory references without timing [22] or using a full trace of executed instructions for relatively fast simulation with complete fidelity [3]. Reducing traces has been considered important for practical reasons of storage space and simulation speed. For instance, Iyengar et al. [8] defined an "R metric" to guide reducing trace sizes while still maintaining the branch related properties of the original traces. Our work presented in this

Benchmark	Conf. 1	Conf. 2	Conf. 3	Conf. 4	Conf. 5	Benchmark	Conf. 1	Conf. 2	Conf. 3	Conf. 4	Conf. 5
mcf	1.6%	−0.3%	−0.3%	0.2%	−0.1%	*art*	3.4%	−0.9%	−1.0%	0.6%	0.5%
gzip	0.0%	0.0%	0.0%	0.0%	2.5%	*galgel*	−0.5%	−0.6%	−0.5%	0.4%	−0.4%
vpr	−0.8%	0.4%	0.4%	−0.3%	1.7%	*equake*	0.2%	−0.7%	0.3%	−0.2%	0.0%
twolf	2.0%	−0.5%	1.0%	−0.7%	−1.7%	*swim*	0.0%	0.0%	0.1%	−0.1%	0.1%
gcc	−0.7%	2.1%	0.5%	−0.3%	5.3%	*ammp*	−2.2%	1.1%	0.0%	0.0%	1.8%
crafty	0.0%	0.1%	0.0%	0.0%	2.6%	*applu*	−0.1%	0.0%	0.4%	−0.2%	0.3%
parser	3.7%	−0.8%	2.8%	−2.0%	−1.7%	*lucas*	0.0%	0.0%	0.9%	−0.5%	0.0%
bzip2	0.9%	−1.0%	0.2%	−0.2%	−0.4%	*mgrid*	−0.5%	−3.3%	−0.1%	−0.7%	0.1%
perl	0.0%	0.0%	0.0%	0.0%	−4.5%	*apsi*	0.0%	−9.3%	0.5%	−0.4%	0.1%
vortex	0.0%	0.1%	0.4%	−0.4%	0.7%	*fma3d*	0.0%	0.0%	−0.5%	0.3%	0.0%
gap	0.0%	0.0%	9.8%	−9.7%	0.0%	*facerec*	0.0%	0.0%	0.0%	0.0%	0.2%
eon	0.0%	0.0%	0.0%	0.0%	1.5%	*wupwise*	0.1%	0.0%	0.1%	−0.1%	0.3%
						mesa	0.0%	0.0%	−0.1%	0.1%	0.7%
Avg. Error	1.2%	2.0%	2.0%	2.0%	1.7%	*sixtrack*	−0.5%	0.1%	0.0%	0.0%	0.1%

Table 2. Relative performance difference between the pairwise dependent cache miss model and `sim-outorder`. The examined five configurations are identical to the baseline machine configuration (Table 1) except a single parameter. In Configuration 1, the L2 cache is 2MB instead of 1MB ("larger L2 cache"). In Configuration 2, the L2 cache is 512KB instead of 1MB ("smaller L2 cache"). In Configuration 3, the memory latency is 200 cycles instead of 300 cycles ("faster memory"). In Configuration 4, the memory latency is 400 cycles instead of 300 cycles ("slower memory"). In Configuration 5, the L2 hit latency is 20 cycles instead of 12 cycles ("slower L2 cache").

paper has a different goal than previous research: achieving high accuracy and simulation speeds with filtered traces when modeling a superscalar processor system. Our techniques are particularly powerful for building a fast multicore processor simulator where the focus of study is on the system-wide organizational artifacts. There are recent trace-driven multicore simulators that use filtered traces like our work. Zauber [15] used Turandot [17] for collecting single-thread traces and TPTS [7] used Simics [16]. We expect that our techniques can be easily integrated into trace-driven multicore processor simulators like Zauber and TPTS.

In terms of modeling the behavior of superscalar processors, filtered trace-driven simulation is rivaled by analytical and statistical modeling techniques. Karkhanis and Smith [11] have presented an accurate framework for analyzing the performance of a workload on a superscalar processor with a static configuration. Once a workload profile is obtained, their model allows a quick evaluation of the impact of a change of a few processor parameters. However, it requires re-profiling of the workload to change the L2 cache configuration. Noonburg and Shen [18] have proposed a framework for statistical modeling of superscalar processors. While such a model-based approach is extremely useful when considering a few design parameters quickly, it does not diminish the role of fast and accurate simulation methods. Importantly, these analytical and statistical models do not capture the interactions of co-scheduled programs on a multicore processor and hence are unable to predict the impact of such interactions. Recently, Chen and Aamodt [6] improved on the model of [11] by more accurately estimating the CPI component due to long latency data cache misses. Like our work and [11], they consider a sequence of instructions in the "instruction window" to determine the parallelism among the instructions. However, their main focus was on analytically modeling a superscalar processor's performance using a full trace. Our focus in this work is to develop a simulation method that is accurate (with timing and dependence information collected during trace generation) and storage efficient (by using filtered traces).

Cache properties have also been used to reduce memory traces while retaining total accuracy. Using the property of cache inclusion [5], a trace can be filtered of references guaranteed to hit in actual simulation. Wang and Baer [23] use a direct-mapped "filter cache" to filter memory references. Further work by Kaplan et al. [10] has yielded trace reduction techniques *Safely Allowed Drop (SAD)* and *Optimal LRU Reduction (OLR)*, which accurately simulate the LRU policy. These further filter out hits, and OLR is provably optimal for the LRU policy. However, these properties have not been studied in the context of timing accuracy in the context of superscalar processors.

While not directly comparable to our approach, there are other interesting work done to reduce simulation times. MinneSPEC tries to reflect the behavior of SPEC2k while using a smaller, but representative workload [13]. The SMARTS framework [24, 25] determines simulation points that are representative of the behavior of an application. This same technique can be applied to a filtered trace-driven approach during the trace capture phase. Rather than a random sample of simulation points, Sherwood et al. [20] have proposed to analyze the program beforehand to determine which points are most representative.

8. Conclusions

This paper introduced and studied three strategies to assess the impact of a long latency memory access on superscalar processor performance in trace-driven simulations. Compared with a full trace based simulation method that is accurate yet incurs large trace storage and simulation time overheads, our simulation strategies use storage efficient filtered traces that carry enough information to compute latencies between trace items at trace simulation time. Previous simulation methods that use memory traces, both filtered and unfiltered, have not attempted to model superscalar processor performance accurately. We found that the pairwise dependent cache miss model robustly gives the smallest errors (4.8% RMS) among the three strategies we examined for a 4-issue processor model. Compared with detailed execution-driven simulation, our technique achieves a simulation speedup of 30.7× on average when running the SPEC2k benchmark suite.

As current and future processor research is centered on the idea of multicore architectures, the importance of studying system-wide resources such as shared L2 cache, on-chip interconnection network and memory controller, will continue to grow. Our study forms a basis for tools that enable fast and accurate multicore simulations at an early design stage using a set of pre-determined superscalar processor core configurations as "plug-in" components rather than a target of study.

This paper focused on how the out-of-order instruction execution capability of a superscalar processor, as a function of the ROB size, affects the trace-driven simulation methodology. In our future research, we will study how to achieve simulation accuracy in the presence of branch prediction and prefetching.

Acknowledgment

This work was supported in part by NSF grant CCF-0702236 and an A. Richard Newton Graduate Scholarship from the 45th Design Automation Conf. (DAC). The authors thank the anonymous reviewers for their constructive comments.

References

[1] T. Austin, E. Larson, and D. Ernst. "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, 35(2):59–67, Feb. 2002.

[2] S. Borkar *et al.* "Platform 2015: Intel Processor and Platform Evolution for the Next Decade," *Technology@Intel Magazine*, March 2005.

[3] L. Barnes. "Performance Modeling and Analysis for AMD's High Performance Microprocessors," Keynote at *Int'l Symp. Performance Analysis of Systems and Software (ISPASS)*, Apr. 2007.

[4] B. Black, A. S. Huang, M. H. Lipasti, and J. P. Shen. "Can Trace-Driven Simulators Accurately Predict Superscalar Performance?" *Proc. Int'l Conf. Computer Design (ICCD)*, pp. 478–485, Oct. 1996.

[5] J. Chame and M. Dubois. "Cache Inclusion and Processor Sampling in Multiprocessor Simulations," *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1993.

[6] X. Chen and T. Aamodt "Hybrid Analytical Modeling of Pending Cache Hits, Data Prefetching, and MSHRs," *Proc. Int'l Symp. Microarchitecture (MICRO)*, pp. 455–465, Nov. 2008.

[7] S. Cho, S. Demetriades, S. Evans, L. Jin, H. Lee, K. Lee, and M. Moeng. "TPTS: A Novel Framework for Very Fast Manycore Processor Architecture Simulation," *Proc. Int'l Conf. Parallel Processing (ICPP)*, pp. 446–453, Sep. 2008.

[8] V. S. Iyengar, L. H. Trevillyan, and P. Bose. "Representative Traces For Processor Models with Infinite Cache," *Proc. Int'l Symp. High-Performance Computer Architecture (HPCA)*, pp. 62–72, Feb. 1996.

[9] M. Johnson. *Superscalar Microprocessor Design*, Prentice Hall, 1991.

[10] S. F. Kaplan, Y. Smaragdakis, and P. R. Wilson. "Flexible Reference Trace Reduction for VM Simulations," *ACM Trans. Modeling and Computer Simulation*, 13(1):1–38, Jan. 2003.

[11] T. Karkhanis and J. E. Smith. "The First-Order Superscalar Processor Model," *Proc. Int'l Symp. Computer Architecture (ISCA)*, pp. 338–349, June 2004.

[12] S. Kim, D. Chandra, and Y. Solihin. "Fair Cache Sharing and Partitioning in Chip Multiprocessor Architecture," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2004.

[13] A. KleinOsowski and D. J. Lilja. "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *Computer Architecture Letters (CAL)*, Vol. 1, June 2002.

[14] P. Kongetira, K. Aingaran, and K. Olukotun. "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, 25(2):21–29, Mar.-Apr. 2005.

[15] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron. "CMP Design Space Exploration Subject to Physical Constraints," *Proc. Int'l Symp. High-Performance Computer Architecture (HPCA)*, pp. 62–72, Feb. 2006.

[16] P. S. Magnusson *et al.* "Simics: A Full System Simulation Platform," *IEEE Computer*, 35(2):50–58, Feb. 2002.

[17] M. Moudgill, J. Wellman, and J. Moreno. "Environment for PowerPC Microarchitecture Exploration," *IEEE Micro*, 19(3), May-June 1999.

[18] D. B. Noonburg and J. P. Shen. "A Framework for Statistical Modeling of Superscalar Processor Performance," *Proc. Int'l Symp. High-Performance Computer Architecture (HPCA)*, pp. 298–309, Feb. 1997.

[19] J. P. Shen and M. H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*, McGraw-Hill, 2004.

[20] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. "Automatically Characterizing Large Scale Program Behavior," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 45–57, Oct. 2002.

[21] Standard Performance Evaluation Corporation. `http://www.specbench.org`.

[22] R. A. Uhlig and T. N. Mudge. "Trace-Driven Memory Simulation: A Survey," *ACM Computing Surveys*, 29(2): 128–170, June 1997.

[23] W.-H. Wang and J.-L. Baer. "Efficient Trace-Driven Simulation Methods for Cache Performance Analysis," *Proc. ACM SIGMETRICS Conf. Measurement and modeling of computer systems (SIGMETRICS)*, pp. 27–36, 1990.

[24] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," *Proc. Int'l. Symp. Computer Architecture (ISCA)*, June 2003.

[25] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. "An Evaluation of Stratified Sampling of Microarchitecture Simulations," *Proc. Workshop Duplicating, Deconstructing, and Debunking (WDDD)*, June 2004.

[26] J. J. Yi, L. Eeckhout, D. J. Lilja, B. Calder, L. K. John, and J. E. Smith. "The Future of Simulation: A Field of Dreams," *IEEE Computer*, 39(11):22–29, Nov. 2006.