MODELING OUT-OF-ORDER SUPERSCALAR PROCESSOR PERFORMANCE QUICKLY AND ACCURATELY WITH TRACES

by

Kiyeon Lee

B.S. Tsinghua University, China, 2006

M.S. University of Pittsburgh, USA, 2011

Submitted to the Graduate Faculty of

the Dietrich School of Arts and Sciences in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy in **Computer Science**

University of Pittsburgh

2013

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Kiyeon Lee

It was defended on

August 12th 2013

and approved by

Sangyeun Cho, Ph.D., Associate Professor at Department of Computer Science

Rami Melhem, Ph.D., Professor at Department of Computer Science

Youtao Zhang, Ph.D., Associate Professor at Department of Computer Science

Alex K. Jones, Ph.D., Associate Professor at Electrical and Computer Engineering

Dissertation Director: Sangyeun Cho, Ph.D., Associate Professor at Department of Computer Science

MODELING OUT-OF-ORDER SUPERSCALAR PROCESSOR PERFORMANCE QUICKLY AND ACCURATELY WITH TRACES

Kiyeon Lee, PhD

University of Pittsburgh, 2013

Fast and accurate processor simulation is essential in processor design. Trace-driven simulation is a widely practiced fast simulation method. However, serious accuracy issues arise when an out-of-order superscalar processor is considered. In this thesis, trace-driven simulation methods are suggested to quickly and accurately model out-of-order superscalar processor performance with reduced traces. The approaches abstract the processor core and focus on the processor's uncore events rather than the processor's internal events. As a result, fast simulation speed is achieved while maintaining fairly small error compared with an execution-driven simulator. Traces can be generated either by a cycle-accurate simulator or an abstract timing model on top of a simple functional simulator. Simulation results are more accurate with the method using traces generated from a cycle-accurate simulator. Faster trace generation speed is achieved with the abstract timing model. The methods determine how to treat a cache miss with respect to other cache misses recorded in the trace by dynamically reconstructing the reorder buffer state during simulation and honoring the dependencies between the trace items. This approach preserves a processor's dynamic uncore access patterns and accurately predicts the relative performance change when the processor's uncore-level parameters are changed. The methods are attractive especially in the early design stages due to its fast simulation speed.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

As higher microprocessor performance is desired, the microprocessor design has evolved and achieved spectacular breakthroughs over the last decades. In the 1990s, the microprocessor performance showed a significant boost with higher clock frequencies through deeper pipelines and advanced microarchitectural techniques, such as out-of-order execution and aggressive branch prediction [31, 67]. Computer architects have introduced architecture innovations to increase the parallelism in various forms—instruction-level parallelism (ILP), memory-level parallelism (MLP), and the thread-level parallelism (TLP)—present in today's microprocessors [31]. ILP is achieved by executing multiple instructions in parallel supported by multiple functional units and the multi-issue capability of a processor. MLP is achieved as an effect of ILP and the capability of the processor's cache subsystem to issue and track multiple outstanding requests to the main memory. In a single-core processor, TLP is realized by executing multiple threads simultaneously on a single multithreaded processor core.

Starting from the decade of 2000, the trend in microprocessor design changed from a single-core processor to a multicore processor architecture. Rather than squeezing the performance out of a single-core processor core, a multicore processor improves the system performance by increasing the total throughput of the system. In multicore architecture, TLP is realized by executing multiple threads simultaneously on multiple processor cores.

As more and more processor cores are integrated in a single chip, the performance of the underlying memory subsystem is critical to achieve high overall performance. More specifically, as the number of processor cores in a chip increases, the contention in the shared resources, such as the interconnection network, the last-level cache, and the memory controller, has a significant and growing impact on the performance of a multicore processor system. Such shared resources are sometimes referred to as "uncore" components [18, 64], distinguished from the processor core components such as the branch predictor and the L1 caches.

Developing a microprocessor system involves a thorough evaluation of the processor performance, including the effect of advanced microarchitectural techniques like branch prediction and out-of-order instruction execution, over several processor design stages. In the early design stages, when the target processor system is not available, computer architects rely on software simulation techniques with abstract performance models or rely on analytical models to quickly explore a large design space and study the design trade-offs. More detailed cycle-accurate execution-driven simulation, which closely models the events that occur in an actual processor, is required in the later design stages as the microprocessor design gets finalized. After the silicon of a processor is available, the performance measurement units are used to measure the performance of the implemented processor system. The performance evaluation of a processor is a major challenge as it requires various tools, methodologies, and experience [4].

1.1 PROBLEM DEFINITION

Software simulation enables one to quickly analyze the behavior of a complex system and to evaluate subtle design trade-offs in a controlled experimental environment. However, despite all the advantages, simulation may be unacceptably slow. Simulating seconds of program execution in real time may entail days of simulation. This slow simulation speed affects the development progress of a new processor design. Hence, improving the simulation efficiency by increasing the simulation speed without sacrificing the simulation accuracy has been a hot research topic in the computer architecture community. It is particularly important to perform a fast and reasonably accurate simulation in early design stages.

Trace-driven simulation is a widely practiced simulation method when the traces are prepared and fast simulation is required [73, 82]. To run a simulation, a trace of interesting processor events[1] need to be generated prior to simulation. Once the trace has been prepared it can be reused multiple times with different machine configurations. Replacing detailed functional execution with pre-captured trace results in a much faster simulation speed than an execution-driven simulation method. Thanks to its high speed, trace-driven simulation is especially favored in early design stages [73]. Unfortunately, the accuracy of trace-driven simulation has often been questioned when modeling a complex processor such as an out-of-order superscalar processor [10]; the static nature of the trace poses challenges when modeling a dynamically scheduled out-of-order superscalar processor[2] [10, 43, 45]. As a result, trace-driven simulation has been typically limited to modeling relatively simple in-order processors, unless a full instruction trace of a program is available.

In contrast, execution-driven simulation is a simulation method used to simulate the behavior of a processor in detail without traces. It simulates the processor in a cycle-by-cycle basis which provides great flexibility to simulate a complex processor and returns accurate simulation results. To model a superscalar processor, it is believed that full tracing and computationally expensive detailed modeling of processor microarchitecture are required [4]. However, this comes with the cost of a long development time and a slow simulation speed.

In this dissertation, accurate trace-driven simulation using *reduced trace* is considered for modeling superscalar processor performance. The reduced trace only includes accesses to the uncore components, a subset of the entire instructions, and summarizes the instructions executed between operations. There are prior trace-driven simulation works using *filtered trace* [73], which is a trace of memory references filtered from the program instruction stream. In this dissertation, the notion of reduced trace is used to represent a trace of accesses to the uncore components obtained by filtering the L1 cache hits.

Filtered trace based simulation is desirable because filtered trace simplifies the complexity of modeling a processor core, obtains results faster, and requires less storage space than trace-driven simulation using full instruction traces. Trace-driven simulation with filtered trace works well for in-order processors. For example, consider the detailed simulation of a

[1] In this dissertation, a trace of a single processor event is denoted as a “trace item”. The size of a trace item depends on the amount of information stored in the trace item. It is usually in the range of several 10’s of bytes.

[2] In this dissertation, the terms “out-of-order superscalar processor” and “superscalar processor” are used interchangeably.

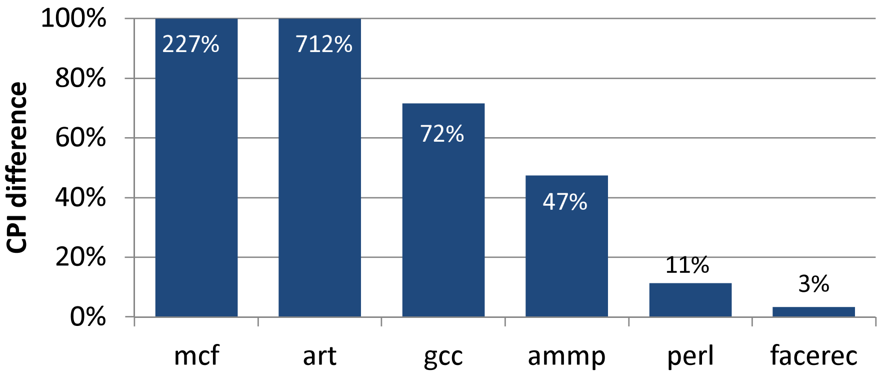


Figure 1: Inaccurate CPI results from a naïve trace-driven simulation of a superscalar processor.

simple in-order processor. During the trace generation phase, one might record the type and the address of every memory operation as well as the number of instructions executed and the number of cycles elapsed since the last memory operation. 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 simulation speed.

However, this straightforward approach does not work for a superscalar processor, since it does not necessarily block during a long latency operation and executes other instructions to hide the latency cost. For example, a superscalar processor executes instructions during a long latency off-chip access to hide the cost of the long latency. Even multiple off-chip accesses can be simultaneously outstanding while the data fetched by individual access is still in transit from the memory. Moreover, the impact of an off-chip access on program execution time is determined dynamically during program runtime and changes with different machine configurations. However, a trace naturally contains the choices made by a core in one particular instance of execution. Figure 1, produced using a typical 4-issue superscalar processor model and a selected set of benchmarks from the SPEC2K benchmark suite [70], shows that using the naïve approach described above to model superscalar processor performance indeed results in very high errors.

It is not straightforward how to assess the impact of a memory access in a superscalar processor with pre-generated filtered trace, especially if one wants to further reduce the amount of trace for faster simulation speed. In this dissertation, practical and effective trace-driven simulation methods are developed and evaluated using reduced trace to model the performance of superscalar processors, especially when the focus of a study is on uncore components such as the L2 cache and the memory controller.

1.2 OVERVIEW OF THE APPROACH

Previously, researchers proposed analytical performance models to quickly derive the performance of superscalar processors [13, 24, 36, 52]. For example, Karkhanis and Smith [36] proposed a first-order analytical performance model to estimate a superscalar processor's performance by paying attention to "miss events" that can stall program execution, such as branch misprediction, instruction cache miss, and data cache miss. The overall performance of a program is derived by adding the ideal CPI and the CPI increase due to the miss events. Chen and Aamodt [13] and Eyerman et al. [24] extended the first order model by improving its accuracy and incorporating more processor artifacts. Michaud et al. [52] built a simple analytical model based on the observation that the instruction-level parallelism (ILP) grows as the square root of the instruction window size. These analytical models derive the overall performance of superscalar processors from relatively simple mathematical models. However, the mathematical models cannot reproduce (or simulate) the dynamic behavior of the processor being modeled. In this dissertation, I focus on trace-driven simulation methods rather than analytical models.

A general trace-driven simulation framework consists of two phases [73, 82] as shown in Figure 2. In the *trace generation* phase, traces are collected from a trace generator. A trace consists of trace items, which may capture every executed instruction of a program, or may contain the information of certain events, such as L2 cache accesses. The trace generator in the figure represents the various tools that can be used for trace generation. The trace generator may include an existing simulator or an emulator that can execute a program

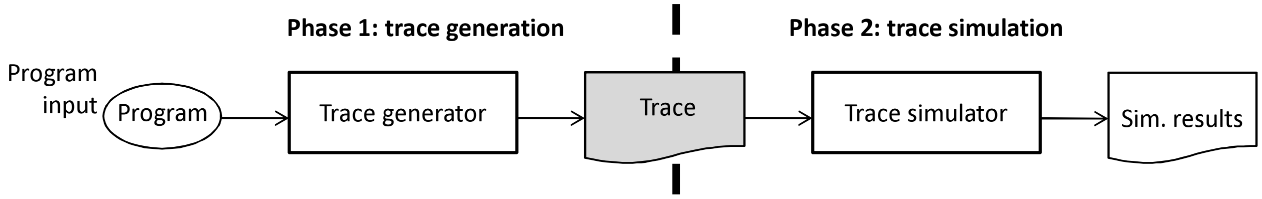


Figure 2: A high-level view of a trace-driven simulation method.

binary [2, 6, 8] or a binary instrumentation tool [49, 57]. In the *trace simulation* phase, the trace simulator exploits the information recorded in the traces. In this work, in the trace generation phase, a cycle-accurate simulator or an abstract timing model on top of a simple functional simulator is used to generate traces. During trace generation, other instructions are filtered and only the L1 cache misses (L2 cache accesses) are traced instead of tracing the entire instructions of a program. Since the trace is a subset of a filtered memory trace, it is named as "reduced trace". In the trace simulation phase, an out-of-order trace simulation is executed by exploiting the information recorded in the reduced traces.

The presented strategies abstract the processor core by replacing the core-level simulation with a reduced trace, and focus on assessing the impact of uncore events on a superscalar processor's performance rather than focusing on the processor's internal events. This dissertation proposes simulation methods to quickly and accurately approximate superscalar processor performance by reasoning about how to treat a cache miss with respect to other cache misses. The trace can either be generated using a cycle-accurate simulator to include the timing information of the processor core, or using an abstract timing model implemented on top of a functional simulator to include an abstract timing information of the processor core. During trace generation, the dependency information between trace items is also recorded. During trace simulation, a trace item is processed considering the information recorded in the trace item. Three simulation models are proposed for trace simulation with timing information—*isolated cache miss model*, *independent cache miss model* [45], and *pairwise dependent cache miss model* [45, 44]—and one trace simulation model is proposed for trace simulation with abstract timing information—*In-N-Out* [43].

The isolated cache miss model computes the impact of each individual L1 cache miss by interleaving the L2 cache hit and miss latency to L1 cache misses during trace generation. Multiple simulation runs are used to skew the alternation of assigning L2 cache hit and miss latencies on L1 cache misses, and compute the L1 cache miss penalty by comparing the number of cycles measured in the same interval. The isolated cache miss model is capable of accurately quantifying the impact of an “isolated” L1 cache miss, however, it is not suitable for a program that frequently creates overlapping L1 cache misses. To accurately model the impact of both isolated and overlapping L1 cache misses, this dissertation proposes the independent cache miss model.

The independent cache miss model determines when an L1 cache miss trace item can be processed by dynamically reconstructing a processor’s reorder buffer (ROB) state during simulation. However, the model is optimistic about when a trace item can proceed because it does not consider the dependency between the L1 cache misses. Effectively, all L1 cache misses are independent, and they do not block the execution of instructions after a miss.

The pairwise dependent cache miss model (PDCM) improves the independent cache miss model by identifying and enforcing the dependency between L1 cache misses. PDCM can model the impact of important processor artifacts, such as instruction caching, branch prediction, L2 data prefetching, and limiting the number of outstanding L2 cache misses using miss status handling registers (MSHRs) [39].

The last model, *In-N-Out*, uses an abstract timing model based on simple functional simulator to quickly generate *reduced in-order traces*. Similar to PDCM, In-N-Out model determines when to process an L1 cache miss trace item by analyzing the ROB occupancy status and honoring the dependencies between trace items. Important processor artifacts like data prefetching and miss status handling registers (MSHRs) can be easily incorporated in the In-N-Out framework. The evaluation results show that In-N-Out produces relatively accurate simulation results with a very high simulation speed.

The experimental results with the SPEC2K benchmark suite demonstrate that the proposed trace-driven simulation models, based on simple yet effective ideas, achieve fast simulation speed and small CPI difference compared with a widely used execution-driven architecture simulator. Among the proposed simulation methods, this dissertation primarily focuses on PDCM and In-N-Out since the two models show the highest simulation accuracy and fastest simulation speed compared to other studied methods. The extensive experiments reveal that, PDCM achieves a very small CPI difference of 3% and fast simulation speeds of 48 MIPS (million simulated instructions per second) on average. In-N-Out achieves a reasonably small CPI difference of 7% and fast simulation speeds of 89 MIPS on average. More importantly, it is observed that PDCM and In-N-Out preserve a processor's dynamic uncore access patterns and accurately predicts the relative performance change when the processor's uncore-level parameters are changed. Compared with a detailed cycle-accurate simulator PDCM and In-N-Out show 55× and 102× simulation speedup on average.

1.3 THESIS CONTRIBUTIONS

Compared with detailed yet slow cycle-accurate simulation methods, the proposed methods have a clear advantage in simulation speed. When compared with a full trace based simulation method, the proposed methods are faster and require smaller storage space. Previous simulation methods that use memory traces, both filtered and unfiltered, have not attempted to model out-of-order superscalar processor performance accurately. In this dissertation, the following contributions are made:

- This work presents practical trace-driven simulation methods employing reduced trace to model the performance of realistic superscalar processors. The methods are practical since they abstract a superscalar processor core's dynamic behavior with high accuracy and only require the timing models for uncore components. The reduced trace can be generated from either a cycle-accurate simulator or an abstract timing model based on a functional simulator or a binary instrumentation tool.
- This work proposed two novel trace-driven simulation methods employing reduced trace, pairwise dependent cache miss model (PDCM) and In-N-Out, to model the performance of realistic superscalar processors. The trace simulation algorithm and the key design issues are discussed, and their effects are quantified.
- Both PDCM and In-N-Out can accurately predict the relative performance of the simulated machine when the machine's uncore parameters are changed. They are also capable

of faithfully replaying how a superscalar processor exercises and is affected by the uncore components.

- The proposed simulation methods are faster than a detailed cycle-accurate simulator. The absolute simulation speed of PDCM and In-N-Out are in the range of MIPS, whereas the simulation speed of a detailed execution-driven simulator is typically in the range of KIPS (kilo simulated instructions per second).

1.4 THESIS ORGANIZATION

The rest of this thesis is organized as follows. Section 2 summarizes related work. Section 3 and Section 4 describe our proposed approaches and present the validation results. Section 5 compares our two signature models: PDCM and In-N-Out. Finally, the conclusion of this work is highlighted in Section 6 and the future research directions are put forth in Section 7.

2.0 BACKGROUND

2.1 OUT-OF-ORDER SUPERSCALAR PROCESSOR

In this dissertation, a machine model is assumed to be a superscalar processor system with two levels of cache memory, L1 cache and L2 cache, and a main memory, as shown in Figure 3. Program instructions and data are separately stored in L1 instruction cache and L1 data cache, respectively. The L2 cache is a unified cache which stores both the program instructions and data. The superscalar processor core model used in this dissertation is sketched inside the dotted box. It has a front-end "fetch pipeline" that fetches instructions from the instruction cache and buffers the instructions for further processing. When a miss occurs in the instruction cache, the L2 cache is accessed to fetch the instructions. If the instruction fetch request also misses in the L2 cache, the main memory is accessed. The instruction fetch bandwidth provides an upper bound on the throughput of all subsequent pipeline stages [67]. It is determined by the instruction cache, branch predictor, and the processor parameters such as the instruction fetch queue size. To achieve sustainable instruction fetch bandwidth, it is important to minimize the branch mispredictions, since modern superscalar processors speculatively execute instructions fetched from a predicted path to increase the instruction-level parallelism (ILP). If the processor mispredicts the path, the processor rolls back to its state that was before the mis-predicted branch, and then executes the instructions fetched from the correct path. Many branch prediction schemes [52, 66] and instruction caching techniques [14, 16, 65] have been developed in the past for high bandwidth instruction fetching.

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

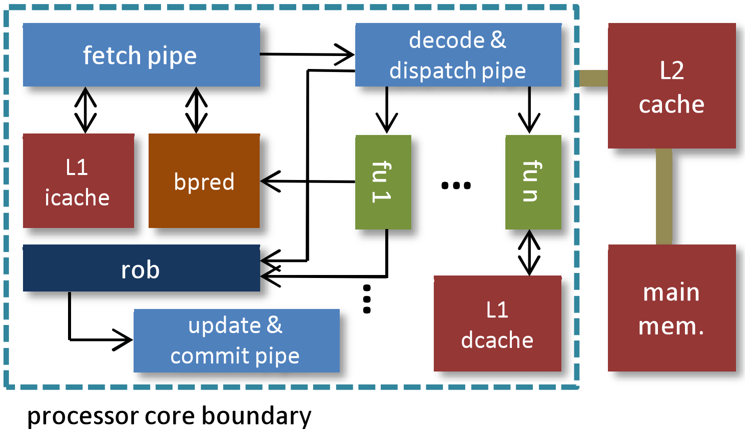


Figure 3: Machine model having a superscalar processor core, L2 cache, and main memory.

in buffers (or *reservation stations*) associated with a specific functional unit until the unit becomes available or until its input operands arrive. Due to the limited number of functional units in a processor, resource conflicts may occur when more than one operations compete for the same functional unit in the same cycle. When an instruction is dispatched, an entry is allocated in the reorder buffer (ROB) so that the "update and commit pipe" can change the architectural state properly in program order as instructions are committed in the presence of special events such as exceptions, branch mis-predictions, and cache misses. The instructions are committed in program order by forcing an instruction to commit only when it becomes the oldest instruction in the ROB (the head of the ROB). Only the instructions in the ROB without any unresolved dependencies are considered to be scheduled at any given time. Hence, the size of the ROB is an important parameter to achieve high instruction-level parallelism or memory-level parallelism. For instance, with a 96-entry ROB, two instructions cannot be simultaneously executed if they are 96 or more instructions away from each other [13, 36]. ROB holds the result of an operation until the associated instruction commits, and provides the result to the depending instructions. For memory instructions, the memory dependency is examined in addition to the data dependency between instructions. When scheduling a load instruction, the store buffer is searched for a preceding store instruction with an unknown memory address. If there is such store instruction, the load instruction cannot be issued until the memory address of the preceding store instruction is calculated.

Because of the disparity between the processor and memory speed [77, 79], the cache subsystem plays a critical role in achieving high performance, particularly for memory-intensive programs, by reducing the number of accesses to the main memory. It exploits the locality presented in a program and stores frequently accessed data to avoid accessing the main memory. The cache subsystem consists of multiple levels of hierarchy, where the upper level caches are faster and smaller than the lower level caches. Modern superscalar processors typically use two or more levels of caches. In this dissertation, the notion of "last-level cache (LLC)" is used to indicate the last level of cache on chip before accessing the off-chip main memory. The processor schedules a memory instruction to a load/store unit which issues a cache access to the L1 data cache. When the cache access misses in L1 data cache, it accesses the lower level caches until it hits in the cache or it reaches the LLC. If the access misses in LLC, it accesses the off-chip main memory. Since the target machine assumes a two-level cache hierarchy, L2 cache is the LLC in the target machine. The L2 cache may be placed inside or outside (uncore) the processor core. The L2 cache that is placed inside the processor core is private to its processor core, whereas uncore L2 cache can be either private or shared among the processor cores. It is noted that the uncore shared L2 cache and main memory are considered as a system-wide resource in multicore processor architectures [23, 38].

To achieve high performance, it is important to reduce the amount of accesses to the main memory and hide the main memory access latency as much as possible due to the long main memory access latency. L2 cache data prefetching may reduce the number of L2 cache misses by speculatively loading the data that are likely to be used in a near future from the main memory to the L2 cache. In this dissertation, a sequential data prefetching technique, tagged prefetch [69], and a stream-based prefetching [71] are employed. The tagged prefetch algorithm uses a tag bit, which is used to mark prefetched blocks that are reused, associated with every cache block. Tagged prefetcher triggers a prefetch request for cache block B + 1, when a cache miss occurs on cache block B, or when a hit occurs on a prefetched cache block B. The stream prefetching algorithm, unlike the simple sequential data prefetching algorithms, monitors the cache access streams and triggers a prefetch request when the cache access is determined to be a part of an identified stream.

To hide the long main memory access latency, the cache subsystem supports multiple outstanding requests to the main memory to overlap the main memory accesses. If two independent main memory accesses occur simultaneously, only the latency of single main memory access will be exposed to the processor. Miss status handling registers (MSHRs) are used to hold the information of an outstanding cache miss until the cache miss is resolved. Hence, the number of outstanding L2 cache misses is determined by the number of L2 MSHRs in the system. An MSHR holds the primary miss and many secondary misses to a cache block. A primary miss is the first cache miss to a cache block and secondary misses are the following cache misses to the same cache block (i.e., delayed hits).

More general description of superscalar processor design and operation can be found in Hennessy and Patterson [31], Johnson [33], and Shen and Lipasti [67].

2.2 PERFORMANCE MODELING METHODS

Before the actual hardware of a target processor system is available, the performance of the target system is estimated using performance modeling techniques. The two most common performance modeling methods are (software) simulation and analytical modeling. It is well known that simulation is accurate than analytical modeling techniques, however, it suffers from long simulation time. On the other hand, analytical modeling techniques are less accurate than simulation, however, they have a clear speed advantage over simulation. Simulation can be classified into execution-driven simulation, which executes actual program instructions, and trace-driven simulation, which is driven from a stored trace file[1]. Table 1 compares three popular performance modeling methods: execution-driven simulation, trace-driven simulation, and analytical modeling.

An execution-driven simulator consists of a functional simulator and a timing simulator. The functional simulator implements an instruction set architecture that can execute a real

[1] Trace-driven simulation may directly use the trace without storing the trace in a disk using on-line tracing techniques. However, this dissertation assumes that trace-driven simulation uses stored traces. On-line method does not incur storage space overheads, however, it does not allow traces to be shared and makes it difficult to obtain repeatable simulation results [40].

Methods		Speed	Required disk space size
Execution-driven simulation	Application-only	Slow	—
	Full-system	Very slow	Small
Trace-driven simulation	Full-trace	Slow	Large
	Filtered-trace	Fast	Moderate
Analytical modeling		Very fast	May require large space

Table 1: Comparing different performance modeling methodologies.

binary. It decodes the instructions in program order to get their operands and store operation results in the target register. The timing simulator, also known as performance simulator, models the processor microarchitecture artifacts. It takes the machine configuration and the decoded instruction information as input and collects various statistics to measure the performance.

There are two types of execution-driven simulation. One is an application-only execution-driven simulation, which simplifies the handling of I/O operations and operating system activities. When simulating a program binary on an application-only execution-driven simulator, the system calls from the binary are emulated by calling the host operating system. Simplescalar [2] is a popular application-only execution-driven simulator suite used in academia. The simulator suite has a fast functional simulator and a detailed timing simulator that models an out-of-order superscalar processor. The other type is a full-system execution-driven simulation, which models the complete hardware system in enough detail to run unmodified operating systems. There are many workloads that require an entire system simulation to obtain meaningful simulation results, such as the database and server workloads [72] and multithreaded workloads [7]. Examples of well-known full-system simulators are SIMICS [50], QEMU [6], gem5 [8], and MARSSx86 [63]. SIMICS and QEMU functionally simulates the entire computer systems, but do not provide modules for timing simulation of processor systems. Hence, SIMICS and QEMU users must develop their own modules to simulate a processor system. An example is GEMS [51], which is a set of modules for SIMICS that models the detailed microarchitecture of multiprocessor systems. On the other hand, gem5 and MARSSx86 simulators provide both the functional simulator and detailed timing model to simulate a processor architecture. Unlike an application-only simulator, a full-system simulator requires few giga-bytes of storage space to keep an image of the disk.

Trace-driven simulation replaces the functional simulation of the execution-driven simulation with pre-generated traces. Once the traces are generated, the traces can be reused many times for timing simulation. The traces may be a full instruction trace or traces of certain events, such as memory references. Conventional trace-driven simulation for superscalar processors employs a full instruction trace to simulate the dynamic behavior of a superscalar processor [62]. However, a full instruction trace requires a large storage space. Moreover, trace-driven simulation with a full instruction trace does not have a clear speed advantage over an execution-driven simulation, since they both simulate the entire instruction stream of a program. On the other hand, filtered-trace simulation requires a much smaller storage space since it only traces specific events rather than the entire instructions of a program. For instance, Dinero IV [11] is a cache simulator that takes memory reference traces and provides the cache hit and miss information. This dissertation focuses on filtered traces that capture only a subset of memory references.

Analytical modeling relies on mathematical equations to model the superscalar processor performance based on several simplifications. It can provide valuable insights in the early processor design stages, and it has a speed advantage over other simulation methods [36]. However, it only provides the estimated performance numbers as the end result, and it cannot reproduce the dynamic behavior a superscalar processor. For accurate performance modeling, analytical models require an instruction trace analysis for each program to obtain the necessary information for their mathematical models.

During processor development, computer architects select appropriate performance modeling methods depending on the purpose and requirement of the modeling work. For instance, to obtain highly accurate simulation results, a detailed cycle accurate execution-driven simulation is used. On the other hand, if the focus of a study is limited to certain events, filtered trace-driven simulation can be used for faster simulation speed. For example, in this dissertation, the processor core is abstracted and filtered trace-driven simulation is used, since the focus of the work is on assessing the impact of uncore accesses on superscalar processor performance.

2.2.1 Trace-driven simulation

Trace-driven simulation is a widely practiced simulation method due to its fast simulation speed and reduced programming effort compared to other detailed simulation methods, such as execution-driven simulation. Trace-driven simulation consists of two phases [73, 82]. 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. The history of trace-driven simulation goes back several decades [69, 73]. In 1966, Belady used trace-driven simulation method to study the replacement algorithms for a virtual storage computer [5].

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. However, accuracy issues arise when using trace-driven simulation for superscalar processors and multicore processors. Black et al. [10] 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. They also determined that sampling techniques present a problem to the accuracy of trace-driven simulation for superscalar processors. Bitar [9] and Goldschmidt and Hennessy [30] discussed the accuracy of trace-driven simulation for multiprocessor studies. Parallel workloads running on a multiprocessor system are likely to introduce timing-dependencies in the memory traces due to the asynchronous interactions between processes (threads), such as dynamic allocation of shared resources and barrier synchronization. The timing-dependencies in trace incur large inaccuracies when trace-driven simulation is used for multiprocessor systems. Nevertheless, Goldschmidt and Hennessy [30] introduced a technique for accurate trace-driven simulation of multiprocessors when timing dependencies are created by locks and barriers.

In previous and current practice, much trace-driven simulation work on memory system simulation has focused on either tracing memory references without timing [73] or using a full trace of executed instructions for relatively fast simulation with complete fidelity [4]. In the meantime, reducing traces has been considered important for practical reasons of storage space and simulation speed. For instance, Chame and Dubois [12] used the property of cache inclusion to filter memory references that are guaranteed to hit in actual simulation and Iyengar et al. [32] defined an "R metric" to guide reducing trace sizes while still maintaining the branch related properties of the original traces. Wang and Baer [74] used a direct-mapped "filter cache" to filter memory references. Further work by Kaplan et al. [35] 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. Agarwal and Huffman [1] proposed techniques to compress traces by exploiting spatial locality. Filtered tracing differs from sampling [68, 80, 81], since filtered trace items are generated throughout the execution of the program. However, previous trace-driven simulation works that use filtered traces have not been done in the context of timing accuracy for modeling superscalar processor performance.

In my previous work [45], I demonstrated that filtered trace-driven simulation can accurately approximate superscalar processor performance. In [45], I introduced three different trace-driven simulation models to approximate the impact of a long-latency memory access on superscalar processor performance with filtered traces: *isolated cache miss model*, *independent cache miss model*, and *pairwise dependent cache miss model*. However, the simulation models require a cycle-accurate execution-driven simulator that models the microarchitecture of the target superscalar processor to generate filtered traces.

In [44], I focused on the pairwise dependent cache miss model (PDCM) among the three strategies introduced in [45]. I presented the trace simulation algorithm in complete detail and discussed the improvements over [45] and many implementation issues. Moreover, I considered important architectural artifacts including: MSHR, data prefetching, and branch predictor. In [43], I proposed an abstract simulation method called In-N-Out to remove such limitation by using a functional cache simulator to generate filtered traces. Since a functional simulator does not provide timing information, In-N-Out resorts to information about data dependency between instructions to estimate the distance between L1 data cache misses.

Filtered trace simulation has been used for experiments with multithreaded parallel applications. Eggers et al. [20, 21] use the memory references of parallel programs to analyze the sharing behavior of parallel applications to evaluate the performance of coherency protocols. In particularly, they analyze the memory reference patterns of write shared data in parallel applications. There are trace-driven multicore simulators that use filtered traces like this dissertation. Zauber [46] and TPTS [42] increase the simulation speed by replacing the core-level simulation with filtered traces. TPTS assumes in-order processor cores and does not model out-of-order superscalar processors. Zauber models out-of-order superscalar processors, however, their work lacked the details of the simulation methodology and evaluation of Zauber. Zauber uses Turandot [55] and TPTS uses Simplescalar and Simics [50] for collecting traces. PDCM and In-N-Out are expected to be easily integrated in such trace-driven multicore simulators.

Finally, traces for single-threaded or multi-threaded programs can be generated using inline tracing technique [21], rather than employing existing simulators. The technique automatically modifies the application binary to insert traps (codes) to collect traces during program execution. Similarly, binary instrumentation tools, such as PIN [49] and Valgrind [57], can be used to quickly generate instruction traces or filtered traces. There are also compiler-based tracing techniques [40] that can reduce the trace generation time and space overhead.

2.2.2 Analytical models for out-of-order superscalar processors

Due to its complexity, evaluating a modern superscalar processor's performance with detailed simulation requires a great deal of efforts on implementing and validating a simulator. This motivated many researchers to develop alternative methods to quickly estimate the performance of a superscalar processor. Hence, much superscalar processor modeling work has focused on building an analytical model. There are four proposals [13, 24, 36, 52] the most related to this dissertation. They all used a functional simulator to generate instruction traces prior to the actual modeling. The collected instruction traces are then profiled to derive the parameters used in their analytical model.

Michaud et al. [52] built an analytical model to study the relations between instruction fetching, branch prediction accuracy, and ILP. In their model, they partitioned the full instruction trace into windows of constant size. They examined the length of the instruction dependency chains in each window, and observed that the length of the longest instruction dependency chain can be used to estimate how many instructions can retire per cycle (IPC) and thereby derive the execution time.

Karkhanis and Smith [36] have presented a first-order analytical model for modeling a superscalar processor performance. Their first-order analytical performance model focuses on "miss events" that can stall program execution, such as branch misprediction, instruction cache miss, and data cache miss. Different equations are introduced to model individual penalties. The overall performance of a program is estimated with the baseline CPI (measured with no miss-events) and the CPI due to these miss events.

Chen and Aamodt [13] extended the first-order model [36] by more accurately estimating the CPI component due to long latency data cache misses. They proposed a method to analytically model the effect of pending data cache hits, data prefetching, and MSHRs, which were not considered in the first-order model. They also demonstrated that their model can estimate the individual effect of data prefetching and MSHRs, and their combined effects as well. Their model considered the effect of pending data cache hits, which was not considered by Karkhanis and Smith. This dissertation also discusses the importance of considering the pending data cache hits and modeling their effect accordingly. Chen and Aamodt assumed perfect branch prediction and instruction caching.

Eyerman et al. [24] proposed a "mechanistic model" which is a revision of the first-order model. They model the execution time between two miss events, namely "interval", and the overall execution time is derived by simply aggregating the execution times of all intervals. For simpler performance penalty formulation, their model focused on the dispatch stage of the processor, whereas the first-order model focused on the instruction issue stage. Similar to Karkhanis and Smith, they modeled branch prediction and instruction caching, but did not model the effect of pending data cache hits, data prefetching, and MSHRs. More recently, Genbrugge et al. [29] proposed "interval simulation", which extends the mechanistic model [24] to raise the level of abstraction for fast multicore simulation. They use a functional simulator to generate a dynamic instruction stream for their model. The significant difference between simulation methods and these analytical models comes from the usage of these models. The goal of using an analytical model is to quickly derive the overall performance

Techniques	Technique description
Sampling	Find the representative simulation points of a program
Workload characterization	Develop an alternative program to replace a large program
Increasing the abstraction level	Focus on a few aspects that are most critical to the performance
Parallel simulation	Achieve simulation speedup by parallelizing a single simulation

Table 2: Various techniques to reduce the simulation time.

number as an end result using mathematical equations, whereas a simulation method is used as a tool to estimate the performance and observe the behavior of a target system.

Noonburg and Shen [58] have proposed a framework for statistical modeling of superscalar processors. Eeckhout et al. [19] compares different simulation methods and advocates the advantage of using statistical simulation than other simulation methods, such as trace-driven and execution-driven simulation, when detailed performance modeling is not necessary. Nussbaum et al. [59, 60] also proposed using statistical simulation for superscalar processors and symmetric multiprocessor system.

While 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 like the ones developed in this research. The significant difference between the simulation methods presented in this dissertation and these analytical models comes from the usage of these models.

2.2.3 Other simulation time reduction techniques

While not directly comparable to the dissertation, there are other techniques to reduce the simulation time, as listed in Table 2. Sampling techniques have been developed to reduce the scope of (detailed) simulation during execution-driven simulation. Sherwood et al. [68] proposed SimPoint, a technique to automatically identify "representative" program intervals that exhibit stable behavior (called "phases"). One could choose to simulate portions of these intervals (e.g., 100M instructions) to predict a benchmark's execution time and other metrics rather than simulate the whole execution, thereby effectively reducing the time needed for simulation. According to [68], each SPEC2k benchmark program has up to 10 phases. Considering that typical SPEC2k benchmarks execute hundreds of billions of instructions, SimPoint has the potential to reduce the amount of detailed simulation by a factor of ~100 (100M per phase x 10 phases / 100B instructions). The average IPC error of SimPoint based simulation, compared with sim-outorder, was reported to be ~3% over the SPEC2k benchmarks.

Another sampling method, SMARTS, was proposed by Wunderlich et al. [80]. Unlike SimPoint that reduces the scope of detailed simulation to specific phases, SMARTS systematically samples program execution intervals with relatively fine granularity without paying attention to program behavior changes. The number of samples and the length of each sample depend on the desired target confidence level. Compared with sim-outorder, SMARTS was shown to achieve 60× simulation speedup and less than 1% error (on average). Compared with SimPoint and SMARTS, the techniques presented in the dissertation resort to cache filtering, a fundamentally different sampling strategy with no bearing on simulation intervals. The proposed methods offer an orthogonal method to speed up detailed simulation itself by focusing on a subset of processor events (cache misses) and abstracting away other details. Naturally, they could work together with either SimPoint or SMARTS (in the context of trace-driven simulation).

Ekman and Stenström [22] used statistical method "matched-pari" comparison to minimize the number of simulation points of a program to achieve certain accuracy. Wenisch et al. [75] presented a sampling framework that replaces "functional warming" with live-points without sacrificing accuracy. Functional warming warms up large microarchitecture building blocks, such as caches and branch predictor, while running functional simulation to quickly move to the next simulation point [80].

Workload characterization is used to develop an alternative program that can replace the long-running benchmark program. One such example is synthetic workloads. Synthetic workloads are not a user program, but they capture the characteristics of the real benchmarks that they wish to represent. Ganesan et al. [27] developed a framework that generates synthetic clones for the target benchmarks using the characterized information of the benchmarks. Genbrugge and Eeckhout [28] improved the statistical simulation methodology by using synthetic trace and an accurate memory data flow model, which models delayed hits, RAW (read after write) memory dependencies, and cache miss correlation. On the other hand, MinneSPEC tries to reflect the behavior of SPEC2K benchmarks while using a smaller, but representative input set [37]. Using this workload, simulation results can be obtained in a reasonable time without developing new simulation techniques. The simulation time is expected to be further reduced, if filtered traces generated with synthetic workloads or MinneSPEC are used in the simulation methods proposed in this dissertation.

There are simple yet effective performance modeling works that achieve fast and accurate performance estimation by increasing the abstraction level. An example is profile-based approaches. Profiled-based performance estimation techniques run in two phases, instrumentation and analysis, similar to that of the trace-driven simulation method. In the instrumentation phase, the instrumentation tool instruments a benchmark to count the number of times each basic block is executed in a specific run. In the analysis phase, the execution time of each basic block in a program are estimated using a simple pipeline simulator. The program execution time is then estimated as $T = \sum_{i=0}^{numBBs} T_i \times Count_i$, where $numBBs$ is the number of static basic blocks in a program, T_i is the time spent to execute basic block i, and $Count_i$ is the number of times basic block i was executed in a specific run [61]. Such approach works well for a simple in-order processor. However, very large errors are shown when the above approach is applied for superscalar processor because it does not consider the key characteristics of a superscalar processor, such as out-of-order issue, long dependency arcs that cross basic block boundaries, etc. The reported error compared with a trace-driven simulator was 125% on average using SPEC95 benchmarks [70]. The large average error can be reduced to 43.4% by using a path tracing technique [3]. Ofelt and Hennessy [61] further reduced the average error of using profile-based performance prediction for superscalar processors to 1.25% by using their "Pairwise Analysis Algorithm (PAA)".

Loh [48] introduced a time-stamping algorithm to model superscalar processor performance using a functional simulator. The time-stamping algorithm concentrates on the time when a particular event can occur rather than simulating the processor cycle by cycle. Fields et al. [25, 26] proposed a method to construct a critical path of a program for microarchitectural performance analysis. The critical path is defined as a sequence of instructions in a program that has the longest cumulative latency. Their work is based on the assumption that the performance of a superscalar processor is mainly determined by the events on the critical path. Chou et al. [15] introduced a simulation method based on their epoch model to quickly derive the memory-level parallelism (MLP) of a program. Their simulator, MLPsim, is a very simplified processor model based on several assumptions. Nonetheless, the simulator shows accurate MLP results, especially when a long off-chip access latency is assumed.

Architects run many simulations in parallel to reduce the overall simulation time. Wenisch et al. [76] proposed a simulation framework that can be parallelized using their checkpoints. Since each checkpoint [75] can be simulated independently, users can run hundreds of simulations in parallel using hundreds of checkpoints on many host machines. Similarly, a single simulation may be parallelized to reduce the simulation time. Mukherjee et al. [56] presented a tool called wisconsin wind tunnel (WWT) II that enables a parallel, discrete-event, direct-execution simulation. In their work, they identified four key operations that can underlie parallel, discrete-event, direct-execution simulation and made alternative implementations of the four key operations in their tool. Assuming a 32-node target machine, WWT II achieves a speedup of $4.1-5.4$ when using 8 host processors. More recently, Miller et al. [53] introduced a distributed parallel multicore simulator infrastructure named Graphite. Using their framework individual simulation is distributed across a cluster of servers to accelerate simulation, which works in a completely transparent fashion to the applications. Graphite achieves near linear speedup on a simulation of a 1000-tile target using from 1 to 10 host machines. Finally, Moeng et al. [54] proposed using Graphics Processing Units (GPUs) to accelerate manycore architectural simulation. They showed that using GPUs for parallel simulation can achieve better scaling with core count then other techniques by implementing a trace-driven many cache simulator using NVIDIA's CUDA toolkit.

3.0 TRACE SIMULATION WITH TIMING INFORMATION

3.1 OVERVIEW

In this chapter, three trace-driven simulation models that use timing-aware traces are introduced. Unlike most previous work that uses a trace-driven simulation framework [73], the notion of *timing* is introduced during the trace generation phase and embed time-related information in the trace. Hence, the trace generator must be able to model the microarchitecture of a processor using a user provided machine definition. Figure 4 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: the intra-core parameters that shape the processor core described in Section 2.1. The *target machine definition* is the system-level processor configuration such as L2 cache configuration and main memory access latency, that completes the overall machine model. Throughout this dissertation, sim-outorder, a detailed out-of-order processor simulator of the SimpleScalar tool set [2], is used to generate traces.

The proposed models use timing-aware *filtered traces*. That is, trace files do not contain all executed instructions during program execution and rather focus on memory access instructions [73]. Moreover, L1 cache hits that do not access the L2 cache are filtered out, further cutting down the number of trace items to store in trace files, similar to [12, 42, 46]. Each trace item in the timing-aware filtered traces captures: (1) the number of executed instructions after the last trace item, (2) the number of elapsed cycles since the last trace item, and (3) the information of the memory instruction that generated the trace item (L1 cache miss): cache access type (data read, data write, or instruction fetch), instruction sequence number, cache address, and write-back address (if a write-back occurred on a cache miss).

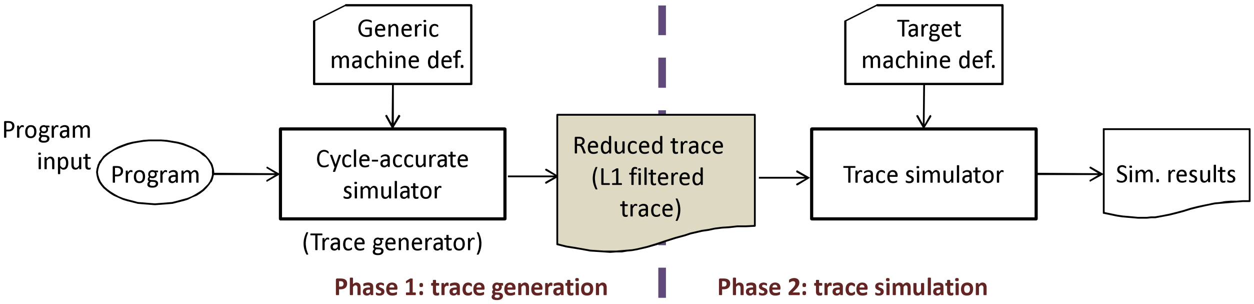


Figure 4: Overall structure of PDCM. It uses a cycle-accurate simulator to generate reduced traces (L1 filtered traces).

In this section, three trace-driven simulation methods are presented and evaluated to quantify the impact of a long-latency memory access in a superscalar processor with timing-aware filtered traces. The strategies are based on three different 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. It is important to note that, in the proposed methods, the processor core configuration used in the trace generator must be identical to the core configuration of the simulated machine. This work assumes that the processor core parameters, such as branch prediction algorithm, ROB size, available functional unit types, and L1 cache configuration, are fixed when the focus of study is on the "uncore" components of a processor chip.

3.2 MODEL 1: ISOLATED CACHE MISS MODEL

3.2.1 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 5(a) captures this idea. Program run 1 has no L2 cache misses, whereas program run

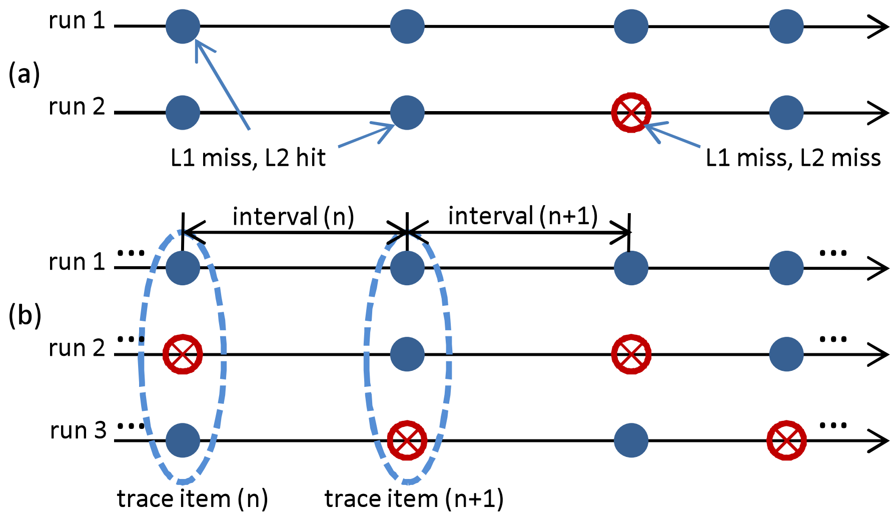


Figure 5: (a) A single "isolated" L2 cache miss in a program run. (b) Using two additional traces generated by interleaving hits and misses to efficiently compute the impact of isolated misses on program execution time.

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 execution time of the program is simply ($T_{\text{run }2} - T_{\text{run }1}$). $T_{\text{run }1}$ can be obtained using a cycle-accurate simulator modeling a perfect L2 cache having a 100% hit rate. $T_{\text{run }2}$ can be obtained by using the same cycle-accurate simulator and giving the L2 cache miss penalty to a specific L2 cache access. One can measure the impact of each and every potential L2 cache miss by repeating this process.

3.2.2 Instruction permeability analysis

While the basic idea of the 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 compute the impact of each potential

L2 cache miss, a technique called *instruction permeability analysis* is used to systematically assign a cache access latency to trace items as they are generated in the trace generation phase. Figure 5(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, the impact of a single L2 cache miss can be computed as would have been done with a trace having only a single L2 cache miss. The configuration sketched in Figure 5(b) is called 2-interleaving because the additional traces have one L2 cache miss every two trace items.

In what follows, we discuss how the impact of a cache miss is analyzed and how such information is associated 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. d_n is defined as $b - a$. Because the nth trace item in trace 2 has a longer latency $(S + L)$ than the latency of the corresponding trace item in trace 1 (S), $b > a$ holds and equivalently $d_n > 0$. Once d_n is obtained, trace item (n) is annotated 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 $L = L'$, the 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 $(b + \eta)$ where $\eta = L' - L$.

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

Dispatch/issue/commit width	4
Reorder buffer (ROB)	64 entries
Integer /Floating point ALUs	4/2
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 3: The baseline machine configuration for evaluating the isolated cache miss model.

miss 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 5(b), the result of the analysis will be a close approximation of what would have been obtained from the exhaustive method. Hence, it is expected 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.

3.2.3 Experimental setup

Table 3 shows the baseline machine configuration that is used to evaluate the isolated cache miss model and the independent cache miss model (in Section 3.3). An ideal instruction cache and a perfect branch predictor are used to isolate the interferences caused by instruction cache misses and branch mispredictions. To evaluate the isolated cache miss model and the independent cache miss model, a selected set of SPEC2K benchmarks is used: *mcf*, *art* (benchmarks with high L1 cache miss rates), *gcc*, *ammp* (with medium L1 cache miss rates), *perl* and *facerec* (with low L1 cache 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. The entire benchmarks in the SPEC2K benchmark suite are used to evaluate the pairwise dependent cache miss model (in Section 3.4) and In-N-Out (in Section 4). The inputs for the entire SPEC2K benchmarks

Integer	Input	Floating point	Input
mcf	inp.in	art	c756hel.in
gzip	input.graphic	galgel	galgel.in
vpr	route	equake	inp.in
twolf	ref	swim	swim.in
gcc	166.i	ammp	ammp.in
crafty	crafty.in	applu	applu.in
parser	ref	lucas	lucas2.in
bzip2	input.graphic	mgrid	mgrid.in
perlbmk	diffmail	apsi	apsi.in
vortex	lendian1.raw	fma3d	fma3d.in
gap	ref.in	facerec	ref.in
eon	rushmeier	wupwise	wupwise.in
		mesa	mesa.in
		sixtrack	inp.in

Table 4: Inputs for the SPEC2K benchmarks.

are listed in Table 4.

The SPEC2K benchmarks used in this dissertation were compiled using the Compaq Alpha C compiler (V5.9) with the `-O3` optimization flag. For each simulation, the initialization phase of the target program [68] is skipped, then caches are warmed up for 100M instructions. The next 1B instructions are simulated after warming up the caches. To evaluate studied simulation methods, *CPI (cycles per instruction) error* is used 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.

3.2.4 Evaluation result

After running experiments, I learned that it is 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 different cache access latencies are assigned to different trace items. Certain trace items occur in one trace file, but not in others, thus mis-aligning the trace items that follow. The number of trace items that are correctly annotated using the 2-interleaving configuration ranged from 23% (*mcf*) to 78% (*perl*). While devising better trace item annotation methods is certainly an interesting question, the presentation in this section is limited 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, 54% (*mcf*), 80% (*art*), 90% (*gcc*), 92% (*ammp*), 95% (*perl*), and 99% (*facerec*) of the trace items were annotated.

Despite being able to considerably reduce the magnitude of CPI errors compared with the naïve method, Figure 6(a) shows that the isolated cache miss model was unable 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, which result in incorrect accumulation of cache miss penalties. Figure 6(b) shows that the studied programs have many L2 cache accesses in short intervals, confirming the 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 the aggressive ad hoc trace item annotation. In an effort to annotate as many trace items as possible from "mis-aligned" trace files, a search-based trace item matching algorithm is employed, which 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.

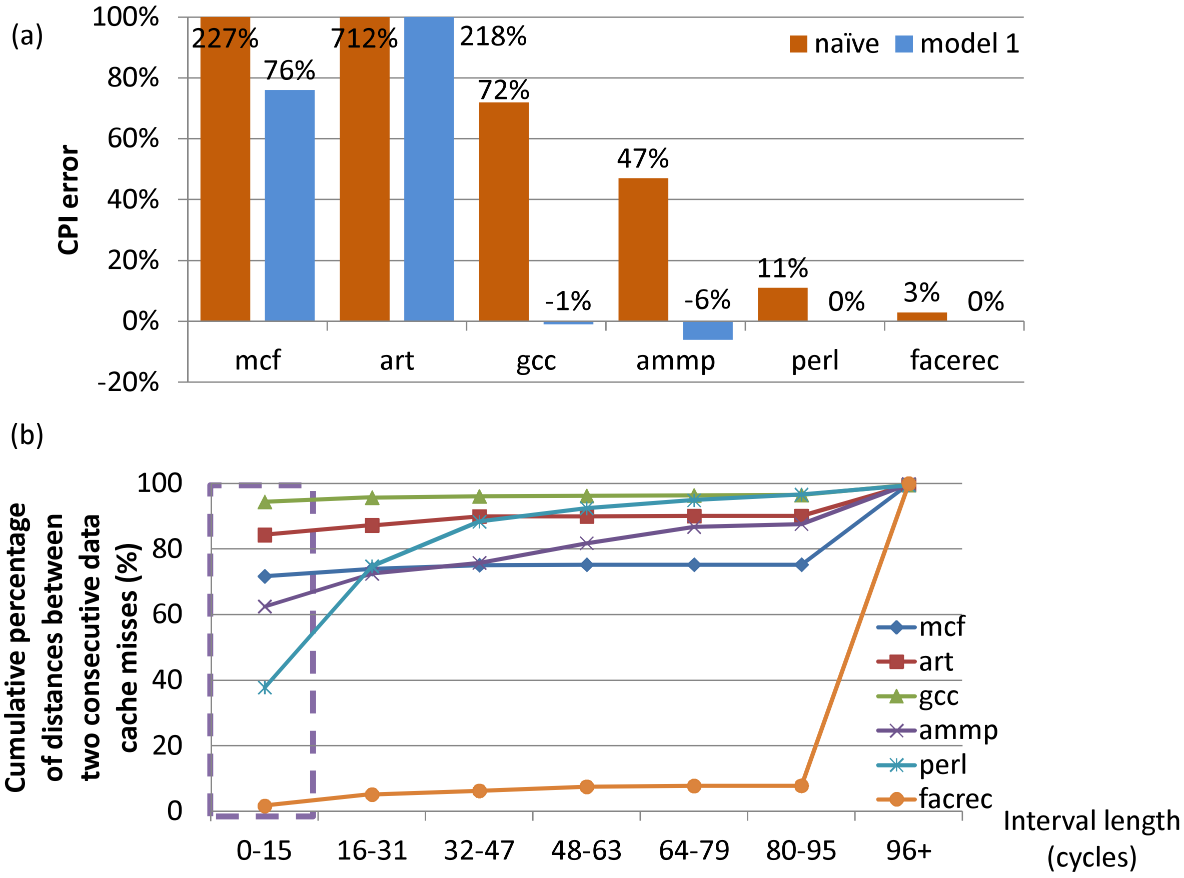


Figure 6: (a) CPI error of the isolated cache miss model. (b) The cumulative percentage of L1 data cache miss intervals in terms of clock cycles. All L1 data cache misses are assumed to hit in the L2 cache. The dotted box shows that there are potentially many independent L2 cache accesses.

3.3 MODEL 2: INDEPENDENT CACHE MISS MODEL

3.3.1 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 in a real superscalar processor.

A Superscalar processor dynamically selects and executes multiple instructions. As a result, more than one cache miss can be outstanding simultaneously. However, there is a limit on the number of pending L1 cache misses, given a processor's limited hardware data structures and inherent dependencies between L1 cache misses. For example, the processor configuration in Table 3 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, or if one depends on the other.

The *independent cache miss model* builds on this observation. It reconstructs the ROB during trace simulation to process a trace item only after the trace item enters the ROB, however, the model ignores the dependency between the trace items in the ROB. In a nutshell, the isolated cache miss model analyzes the ROB occupancy status to determine the progress of trace simulation and when each trace item can issue a cache access.

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. The work focuses on ROB among the many processor data structures based on experiments and a previous analytical performance modeling work done by Karkhanis and Smith [36].

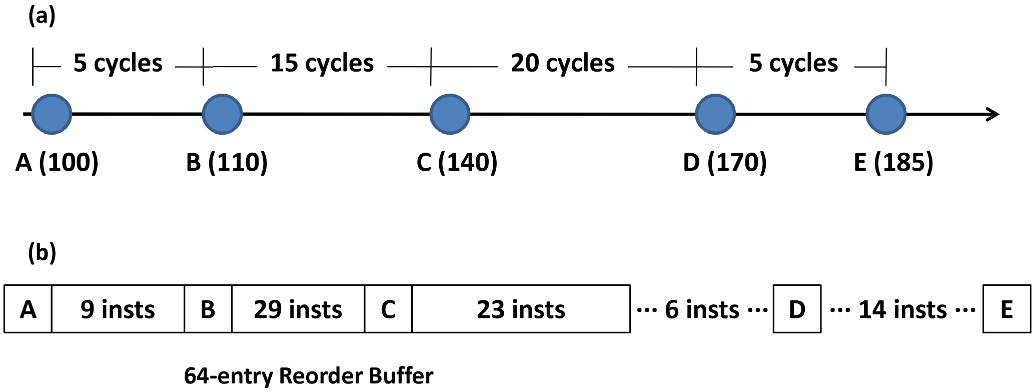


Figure 7: (a) Five trace items (A, B, C, D, and E) recorded in the trace file with timing information in the trace generation phase. Inside parentheses are the instruction sequence numbers. (b) The status of the ROB: Only the first three trace items are in the ROB.

3.3.2 ROB occupancy analysis in the independent cache miss model

The ROB occupancy analysis examines the ROB occupancy status to determine the progress of the trace simulation. More specifically, the trace simulation is continued if the difference between the instruction sequence number of a trace item and the head of the ROB is smaller than the ROB size. Let us turn to the example in Figure 7.

In the example, suppose all five trace items (i.e., L2 cache accesses) miss in the L2 cache and go to the main memory. Given their instruction sequence number, both B and C can be placed in the ROB with A, since the number of instructions between A and C is smaller than the ROB size. However, the number of instructions between A and D is larger than the ROB size. Consequently, D and E cannot issue a cache access while A is in the ROB.

Since all L1 cache misses are assumed to be independent in this method, the L2 cache accesses from B and C can be processed in parallel with the L2 cache access from A. The recorded timing information in the trace items are used to determine the distance between two trace items in the ROB. For example, B and C are processed 5 and 20 (5 + 15) cycles after processing A, respectively. After A returns from L2 cache, A commits and exits the ROB. After A commits, the issued instructions between A and B are committed, which allows the instructions in front of D, as well as the instruction in D, to advance to the ROB.

However, the available entries in the ROB 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 follow and commit at the processor's commit rate. 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 is done in the trace simulation phase. Therefore, the independent cache miss model does not require any trace analysis before simulation. Note that the multiple traces must be generated and analyzed prior to trace simulation in the isolated cache miss model. The details of out-of-order trace simulation algorithm with the ROB occupancy analysis are described in Section 3.4.

3.3.3 Evaluation result

To evaluate the independent cache miss model, the same machine configuration, benchmarks, and the evaluation metric used to evaluate the isolated cache miss model (described in Section 3.2.3) are employed.

Figure 8 compares the CPI collected from `sim-outorder` and the independent cache miss model. The results show that the CPIs observed by the independent cache miss model are in general smaller than the CPIs shown by `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. The largest improvement was shown by *art* compared to the naïve method because the majority of its L1 cache misses occur very closely to each other, as indicated in Figure 6(b). This suggests that there are potentially many independent L2 cache misses in *art* which are accurately quantified using the independent cache miss model.

In the case of *mcf*, a large simulated execution time deviation was observed. This large magnitude of error is attributed to the memory access pattern of *mcf*–there are many trace items that are dependent on other trace items. The profiling results of *mcf* reveals that 79%

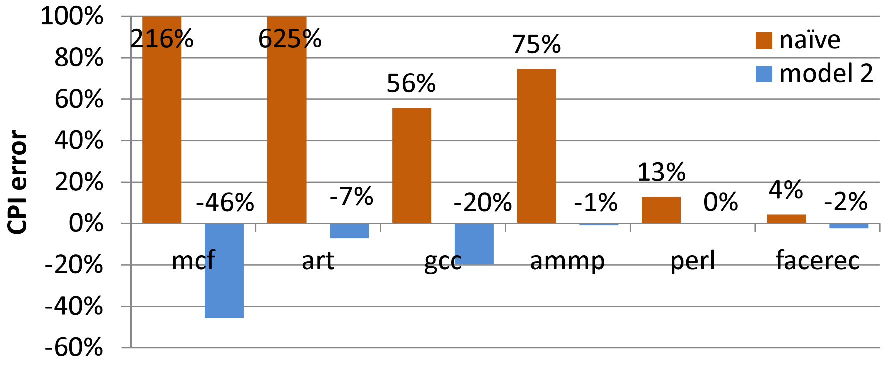


Figure 8: The CPI errors of the independent cache miss model when the ROB size is 64.

of *mcf*'s trace items have dependencies which are neglected in the independent cache miss model. *mcf* shows 0% CPI error when the ROB size is reduced to 4, because the memory accesses that depend on the previous memory accesses are often not placed in the ROB together.

3.4 MODEL 3: PAIRWISE DEPENDENT CACHE MISS MODEL (PDCM)

3.4.1 Basic idea

The independent cache miss model discussed in Section 3.3 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 3.2 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, yet another model is proposed, which combines the strengths of the two previous models. The new model exploits the parallel scheduling capability of the independent cache miss model as well as the 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 (PDCM) honors the dependencies between trace items. To do so, in the trace generation phase the dependencies between trace items are detected and recorded in trace items. 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 dependent cache access is not processed immediately.

3.4.2 Preparing reduced trace in PDCM

Identifying all data dependencies between trace items is critical for accurate filtered trace simulation. In trace generation, to detect the dependencies between trace items, data dependency chains are constructed during trace generation. In the dependency chain, the instruction sequence number of a parent trace item is propagated to the dependent trace items. In this work, the dependency chains already implemented in the modified `sim-outorder` simulator, which is employed for trace generation, is used. Note that a single trace item may depend on multiple trace items. However, the experiments show that storing more than a single ancestor does not produce significantly better results and thus, only one ancestor or none (no dependence) is stored in each trace item. In the presence of multiple ancestors, a heuristic is used to choose the latest ancestor in the instruction sequence, the closest to the trace item under consideration.

Besides the explicit dependency between trace items, there also exists an implicit dependency due to *delayed hits*. A delayed hit occurs when a memory instruction accesses a cache block that is still in transit from the lower-level cache or the main memory. Consider an L1 data cache miss that depends on an L1 delayed hit. This L1 data cache miss must be processed after the previous L1 data cache miss that created the delayed hit, since there exists an implicit dependency between the two L1 data cache misses via the L1 delayed hit in between [13]. To expose all dependencies between trace items during trace simulation,

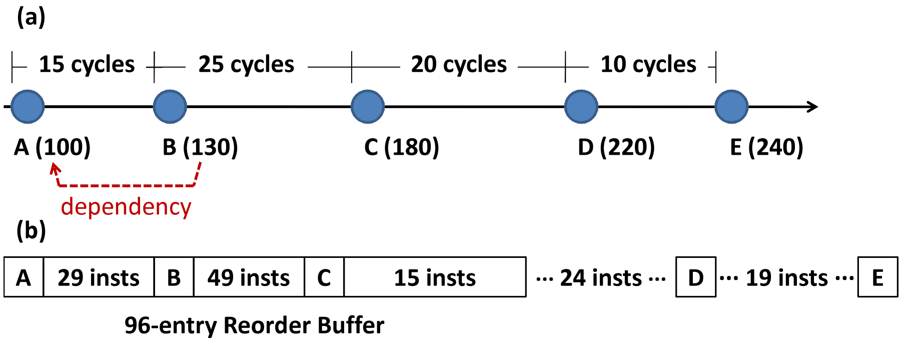


Figure 9: (a) Five trace items (A, B, C, D, and E) recorded in the trace file. Trace item B depends on trace item A, while all other trace items are independent of each other. Inside parentheses is the instruction sequence number assigned to each trace item in program order. (b) The status of the ROB: Only the first three trace items are in the ROB.

trace items are generated for L1 delayed hits as well as L1 data cache misses. To identify delayed hits during trace generation, when a cache block is brought into the cache on a miss, the cache block is marked with the instruction sequence number of the memory instruction that generated the miss. A hit in L1 data cache is assumed is a delayed hit, if the difference between the instruction sequence number of the corresponding memory instruction and the recorded instruction sequence number in the cache block is smaller than a specified range.

3.4.3 ROB occupancy analysis in PDCM

In Section 3.3, the ROB occupancy analysis used in the independent cache miss model is introduced with an example depicted in Figure 7. In PDCM, the ROB occupancy analysis is improved by considering the dependency between trace items. In the independent cache miss model, a trace item can be processed if it is in the ROB. However, in PDCM, a trace item in the ROB cannot be processed if it has an unresolved data dependency.

Let us turn to the example in Figure 9. Suppose all five trace items A, B, C, D, and E miss in the L2 cache, and A is the head of the ROB. Given their instruction sequence number, both B and C can be placed in the ROB with A, since the number of instructions between A and C is smaller than the ROB size. However, the number of instructions between A and D is larger than the ROB size. Consequently, D and E cannot issue a cache access while A is in the ROB. C can issue a cache access in parallel with A, since C is in the ROB and does not depend on A or B. However, B has to wait until the cache access from A is done because it depends on A. After A returns from L2 cache, B can issue a cache access and A commits and exits the ROB. The issued instructions between A and B also commit at the processor's commit rate, which allows the instructions between the tail of the ROB and D, as well as the instruction in D, to advance to the ROB. Meanwhile, E cannot yet enter the ROB. When B commits, the instructions between B and C will follow and commit. As more and more entries become free, E will finally move into the ROB and be issued.

Unlike the independent cache miss model, in PDCM, trace items are generated when a delayed hit occurs. These delayed hit trace items help us correctly analyze the ROB occupancy status. Assume that there is a memory instruction with instruction sequence number 120, and it issues a cache access after trace item B. If the cache access goes to the same cache block as B, a delayed hit will occur. In such case, trace item D cannot enter the ROB after trace item A commits, because the memory instruction 120 becomes the head of the ROB and the distance between the memory instruction 120 and D is larger than the ROB size.

In essence, the ROB occupancy analysis monitors the ROB occupancy after each successive trace item, allows all L2 cache accesses without data dependency stalls in the ROB to issue, and blocks any further processing of the following trace items if the ROB is full.

3.4.4 Modeling a superscalar processor

Table 5 lists the notations used in this section.

3.4.4.1 Reconstructing the ROB During trace simulation, the ROB is reconstructed with a linked-list referred to as *rob-list*. The trace items fetched from a trace file are inserted in *rob-list* and sorted in increasing order of their instruction sequence number (*ISN*). The trace item with the smallest *ISN* in *rob-list* becomes the head of the ROB (*robHead*).

If a trace item has an *ISN* that is greater than or equal to the sum of the ROB size and the *ISN* of *robHead*, the trace item cannot enter the ROB. However, since the instructions

current_time	The current clock cycle time
ISN	The instruction sequence number
rob-list	The list used to reconstruct the ROB
robHead	The trace item in the head of *rob-list*
issue-list	The list used for out-of-order trace simulation
issueHead	The trace item in the head of *issue-list*
robReadyTime	The time when a trace item can be processed if it has no data dependency stalls
traceProcessTime	The time to process a trace item
resolveTime	The time when the cache access from a trace item is done

Table 5: Notations used in Section 3.4.4.

are issued out of order during trace generation, the trace items in a trace file are not written in program order. Hence, when ROB is determined to be full, the trace items that can enter the ROB may not have been fetched from the trace file. To capture the correct ROB occupancy status, trace items are fetched until the difference between a trace item's *ISN* and *robHead*'s ISN is larger than a specified range. The size of the range does not affect the simulation accuracy, but it should be larger than the ROB size in order to fetch all the trace items that can enter the ROB. In the experiments, trace items were fetched until the difference was larger than two times the ROB size. The trace items that cannot enter the ROB are marked as "pending" trace items in *rob-list*. For instance, in the ROB example in Figure 9, trace items D and E are pending trace items when A is *robHead*. When *robHead* commits, the pending trace items can enter the ROB if there is enough room left in the ROB. New trace items are fetched from the trace file if there are no pending trace items.

3.4.4.2 Out-of-order trace simulation The time to process a trace item (*traceProcessTime*) is determined based on the ROB occupancy analysis, the recorded cycle count, and the dependency information. If a trace item has a parent trace item, the trace item has to wait until its parent's *resolveTime* is known. Otherwise, the ROB occupancy status is analyzed and the recorded cycle count is exploited to estimate when the trace item can be processed (*robReadyTime*). Hence, *traceProcessTime* of a trace item is the larger of *robReadyTime* and the parent's *resolveTime*. After *traceProcessTime* of a trace item is estimated, the trace item is inserted in a linked-list that which is denoted as *issue-list*. *issue-list* sorts the trace items in increasing order based on their *traceProcessTime*. In trace simulation, the trace item in the head of *issue-list* (*issueHead*) is processed, which has the smallest *traceProcessTime*. After *issueHead* is processed, *issueHead* is removed from *issue-list* and the next trace item in the list becomes the new *issueHead*.

Figure 10 illustrates step by step how the trace items in the example of Figure 9 are handled by the ROB occupancy analysis. The first row in the figure shows *rob-list* and *issue-list* with trace items A, B, C, and D. Assume trace item A is *robHead* and A's *traceProcessTime* is (cycle) N. When trace items B and C are inserted in *rob-list*, their *robReadyTime* is set to N + 15 and N + 40, respectively. Since C does not have a parent trace item, C's *traceProcessTime* is the same as *robReadyTime*. However, B depends on A, hence, B's *traceProcessTime* cannot be estimated until A's *resolveTime* is known. Consequently, only C is inserted in *issue-list*. Note that D is a pending trace item when A is *robHead*.

At cycle N, A is processed and A's *resolveTime* is set to N + memory access latency. After processing A, A is removed from *issue-list* and C becomes the new *issueHead*. Since A's *resolveTime* is now known, B's *traceProcessTime* is computed and B is inserted in *issue-list* as shown in the second row.

There are two approaches—*eager* and *lazy*—when estimating the dependent trace item's *traceProcessTime*. The eager approach processes a dependent trace item immediately after its parent trace item is resolved. On the other hand, the lazy approach delays the processing of the dependent trace item by the number of cycles between the parent and the dependent trace item. The rationale is that there may be other instructions depending on the parent trace item and executed before the dependent trace item. Both approaches were studied and the results showed that the lazy approach achieves higher accuracy on average than the eager approach. The experiment results using both approaches is shown in Section 3.4.6.1.

Continuing with the example, C is processed in cycle N + 40, and C's *resolveTime* is set to N + 40 + memory access latency. After processing C, C is removed, and B becomes the new *issueHead*. At cycle N + memory access latency, A is removed from *rob-list*, and B becomes the new *robHead*. The pending trace item D can now enter the ROB as shown in the last row of the figure.

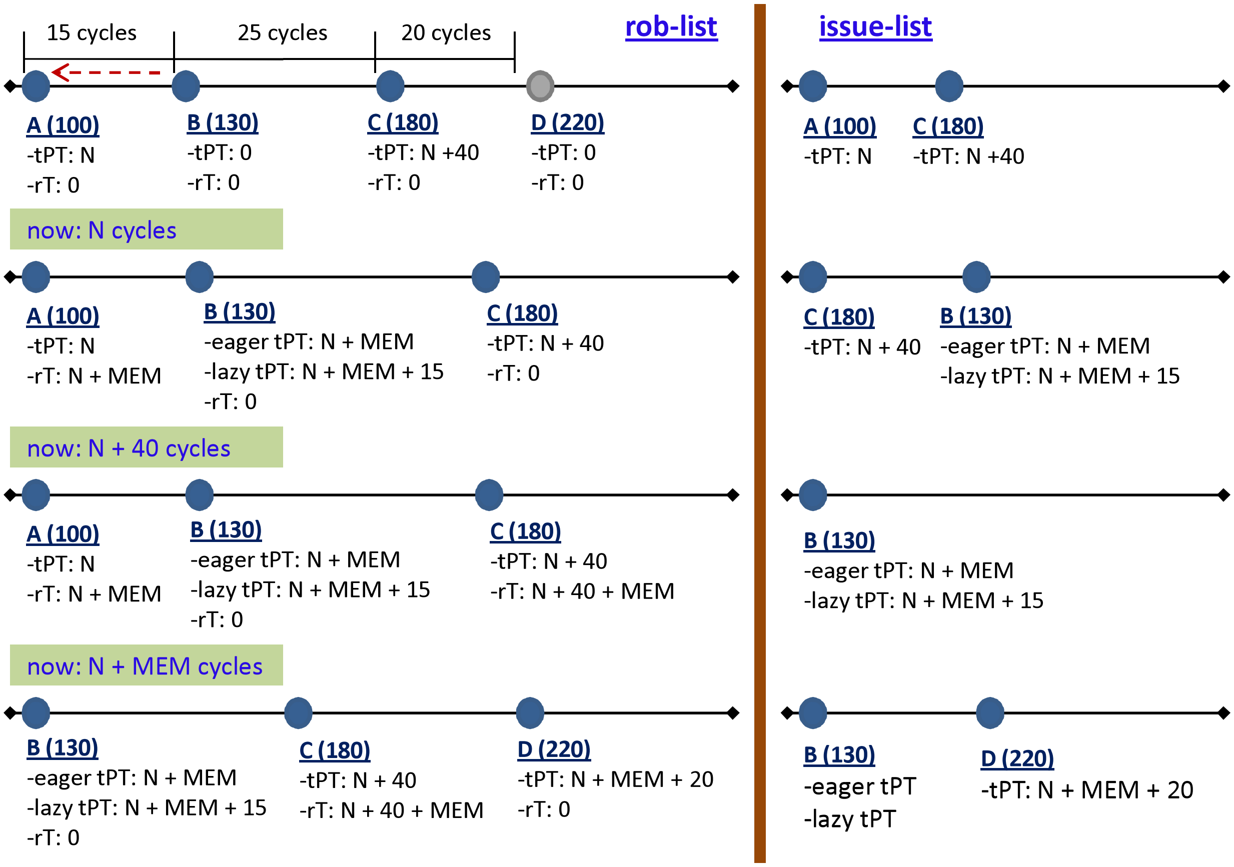


Figure 10: Using *rob-list* and *issue-list* to reconstruct the ROB and issue L2 cache accesses out of order during trace simulation. The example builds on the ROB example in Figure 9. The red dotted arrow shows that trace item B depends on A. The cycle counts between trace items on the first row assume perfect L2 cache. Pending trace item D in the first row is depicted with lighter color. Several abbreviations are used in the example. "tPT" represents *traceProcessTime*, "rT" represents *resolveTime*, and "MEM" is the memory access latency. "eager tPT" and "lazy tPT" stand for eager and lazy estimation of *traceProcessTime*, respectively.

```
 1: while (1) do
 2:   if (current_time >= next_commit_time) then
 3:     ReconstructROB();
 4:   end if
 5:   while (current_time == next_event_time) do
 6:     ProcessTraceItem();
 7:   end while
 8:   if (END_OF_FILE) then
 9:     return;
10:   end if
11: end while
```

Figure 11: High-level pseudo-code of trace simulation in PDCM. *next_commit_time* indicates the time to reconstruct the ROB and *next_event_time* is the time to process the trace items.

Now that I described how the two key ideas of PDCM—ROB occupancy analysis and out-of-order trace simulation—are implemented, let us now move on to the details of the trace simulation algorithm.

3.4.4.3 Simulation algorithm of PDCM Figure 11 shows the main loop in the trace simulation algorithm. PDCM operates in two major steps: (1) reconstructing the ROB (line 3) and (2) processing the scheduled trace items (line 6). The details of each step are described below. The pseudo-codes presented in this section contain a dot (.) notation to represent the association between a trace item and the recorded information. For example, *robHead.ISN* means *ISN* of *robHead*.

• **ReconstructROB()**: Figure 12 describes how the ROB is reconstructed in trace simulation. ROB is reconstructed by removing *robHead*s and inserting pending trace items or new trace items fetched from the trace file. When the simulated clock cycle time (*current_time*) reaches the time to commit a trace item (*next_commit_time*), the algorithm attempts to remove *robHead* from *rob-list* (lines 2–9). If *robHead*'s *resolveTime* is larger than 0 and *current_time* is larger than *robHead*'s *resolveTime*, *robHead* commits and the next trace item in *rob-list* becomes the new *robHead*. If *robHead* is a write trace item, a write access to L2 cache is issued from *robHead* before it is removed (line 3–5). The trace items are committed until either *rob-list* becomes empty or the new *robHead* is not ready to commit. After committing the trace items, *rob-list* accepts "pending" trace items (lines 10–15). If there is no pending trace items and if the ROB is not full, new trace items are fetched from the trace file (lines 16–26). After a trace item is inserted in *rob-list*, the trace item is inserted in *issue-list* if the trace item's *traceProcessTime* can be computed.

• **ProcessTraceItem()**: The L2 cache is accessed by *issueHead*, as described in Figure 13. The L2 cache access latency is used to set *resolveTime* of the corresponding node in *rob-list* (lines 3 and 4). After *issueHead* accesses the L2 cache, *rob-list* is searched to find the dependent trace items. The identified dependent trace item's *traceProcessTime* is computed, and the dependent trace items are inserted in *issue-list* (lines 6–12). After processing *issueHead*, the next trace item in *issue-list* becomes the new *issueHead* (line 16), and the algorithm determines when to process the new *issueHead* (line 17).

If *issueHead* is a write trace item, an access to the L2 cache does not occur, but the algorithm sets *issueHead*'s *resolveTime* to *current_time*. The write trace item accesses the L2 cache when it commits from *rob-list*.

```
robNode = NULL;
while (robHead is not NULL) and (robHead.resolveTime $>$ 0) and (current_time $>$ robHead.resolveTime) do
  if (robHead is a write trace item) then
    Issue a write access to L2 cache;
  end if
  robNode = robHead.next; /*next trace item in rob-list*/
  Commit robHead; /*remove robHead from rob-list*/
  robHead = robNode;
end while
while (robNode is not NULL) and
(robNode.ISN $-$ robHead.ISN $<$ ROB size) do
  if (robNode.pending == TRUE) then
    robNode.pending = FALSE; /* insert pending trace items in rob-list */
  end if
  robNode = robNode.next;
end while
while (1) do
  newTrace = a new trace item fetched from the trace file;
  if (newTrace.ISN $-$ robHead.ISN $<$ ROB size) then
    insert newTrace in rob-list;
  else if (newTrace.ISN $-$ robHead.ISN $\geq$ ROB size) and
  (newTrace.ISN $-$ robHead. ISN $<$ $2\times$ ROB size) then
    newTrace.pending = TRUE;
    insert newTrace in rob-list;
  else
    break;
  end if
end while
```

Figure 12: High-level pseudo-code for reconstructing the ROB.

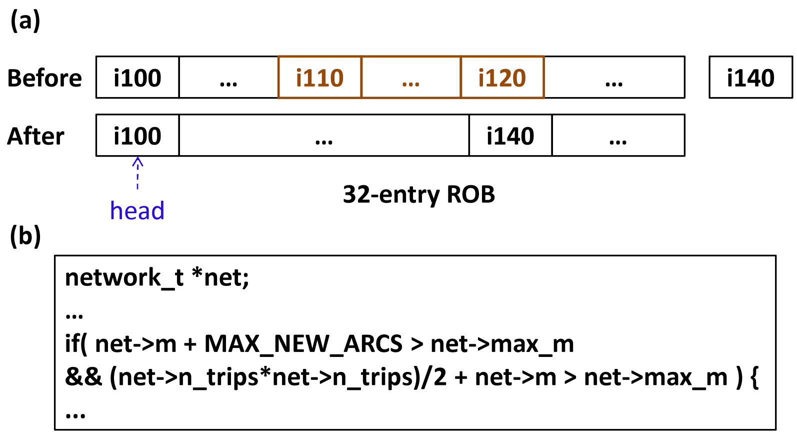
```
rob = issueHead's corresponding trace item in rob-list;
if (issueHead is a read trace item) then
    make read access to L2 cache;
    rob.resolveTime = current_time + L2 access latency;
    node = robHead;
    while (node is not NULL) do
        if (node's parent.ISN == issueHead.ISN) then
            node.traceProcessTime =
                MAX(node.robReadyTime,
               current_time + L2 access latency + elapsed cycles between issueHead and node);
            Insert node in issue-list;
        end if
        node = node.next; /*next trace item in rob-list*/
    end while
else
    rob.resolveTime = current_time; /* IssueHead is a write trace item*/
end if
issueHead = issueHead.next; /*next trace item in issue-list*/
next_event_time = issueHead.traceProcessTime;
```

Figure 13: High-level pseudo-code for processing a trace item.

3.4.4.4 Modeling various processor artifacts in PDCM

The algorithm described above can be easily extended to model important processor artifacts, such as branch mis-predictions, instruction caching, MSHRs, and data prefetching. To add new processor artifacts in the analytical models [13, 24, 36, 52], the constructed mathematical equations are revised or new equations may be required. This can be a burden when new machine configurations need to be modeled. Unlike analytical models, PDCM does not rely on mathematical equations. The processor artifacts can be modeled with a little programming effort—revising the trace generator or the trace simulator. For instance, the effect of branch misprediction is modeled by simulating a branch predictor during trace generation and L2 data prefetching is modeled by implementing a data prefetcher in the trace simulator.

• **Modeling branch prediction**: To model the branch prediction, a realistic branch predictor is employed in the trace generator. Branch mis-predictions during trace generation create "speculative" trace items when the program is executing on the mis-predicted control paths. In trace generation, the speculative trace items are distinguished from the non-speculative



```
network_t *net;
...
if( net->m + MAX_NEW_ARCS > net->max_m
&& (net->n_trips*net->n_trips)/2 + net->m > net->max_m ) {
...
```

Figure 14: (a) The ROB occupancy status before and after the branch mis-prediction is resolved. Assume instructions 110 through 120 are fetched from an incorrectly predicted control path. (b) An example from *mcf* where a branch depends on memory instructions.

trace items, and the dependency information is collected only if the parent trace item is a non-speculative trace item. In trace simulation, a speculative trace item accesses the L2 cache as a realistic superscalar processor would do. However, the speculative trace item is removed from *rob-list* after it is processed.

In this research two important aspects that can affect the accuracy of branch handling are observed. First, speculative trace items can affect the ROB occupancy analysis, and second, a branch instruction depending on the data brought by a memory instruction can affect the estimation of *traceProcessTime*. Let us turn to Figure 14 for illustration.

In Figure 14(a), assume instruction 100 is the head of the ROB and a branch misprediction occurs from instruction 110 allowing the speculative instructions to enter the ROB. After the branch is resolved, the instructions behind instruction 110 are squashed and the processor fills the ROB with instructions fetched from the correct path. Instruction 100 and 140 are in the ROB at the same time even though they are more than ROB size number of instructions away. To handle this case correctly during the ROB occupancy analysis, ISN is incremented only when it is assigned to the non-speculative instructions in trace generation.

The second observation is about the perfect L2 cache assumed in trace generation. If branch instructions depend on the data brought by L2 cache misses, the number of speculative instructions with a perfect L2 cache and a realistic L2 cache will be different. Figure 14(b) shows an example from the *mcf* benchmark in the SPEC2K benchmark suite, where a branch instruction depends on memory instructions. To address this issue, an extra trace item for a branch instruction is generated if the branch depends on a trace item. In trace simulation, the trace items behind a branch trace item in *rob-list* are not processed until the branch trace item is processed. In Section 3.4.6.1, the results show that such approach accurately models the effect of branch mis-predictions.

• **Modeling i-caching**: Finally, to model the effect of instruction caching, a realistic instruction cache is employed in trace generation. If L2 cache is accessed by an instruction cache miss, an L2 cache miss is allowed to occur in trace generation. The timing information is recorded in the trace and exploited in trace simulation.

• **Modeling MSHRs**: In this dissertation, the MSHRs for L2 cache misses are considered; the number of outstanding L2 cache misses is limited by the number of available L2 MSHRs.

Extending PDCM with L2 MSHRs is relatively straightforward. When an L2 cache miss occurs from *issueHead*, the L2 cache miss cannot be processed if there is no available MSHRs. In such case, *issueHead*'s *traceProcessTime* is changed to the time when an MSHR becomes available and reorder *issue-list*.

• **Modeling a data prefetcher**: In this research, a tagged prefetcher [69] is modeled. The tagged prefetcher fetches the next sequential cache block when a miss occurs, or when a hit occurs in a prefetched block. Since the trace items represent the L2 cache accesses, the tagged prefetcher simply needs to monitor the L2 cache accesses from the trace items and make a prefetch request to the memory if necessary.

3.4.5 Experimental setup

Table 6 lists the "baseline" and "realistic" superscalar processor configurations used to evaluate PDCM. The configurations are intended to resemble the Intel Core 2 Duo processor [17]. However, the baseline configuration does not incorporate any processor artifacts. The accuracy of PDCM is first demonstrated with the baseline configuration. PDCM is then

<table>
<tr><th></th><th>Baseline</th><th>Realistic</th></tr>
<tr><td>Dispatch/issue/commit width</td><td colspan="2">4</td></tr>
<tr><td>Reorder buffer</td><td colspan="2">96 entries</td></tr>
<tr><td>Load/Store queue</td><td colspan="2">96 entries</td></tr>
<tr><td>Integer ALUs</td><td colspan="2">4</td></tr>
<tr><td>Floating point ALUs</td><td colspan="2">2</td></tr>
<tr><td>L1 d-cache</td><td colspan="2">2 cycles, 32KB, 8-way, 64B line size, LRU</td></tr>
<tr><td>L1 i-cache</td><td>Perfect</td><td>same as L1 d-cache</td></tr>
<tr><td>L2 cache (unified)</td><td colspan="2">12 cycles, 2MB, 8-way, 64B line size, LRU</td></tr>
<tr><td>Main memory latency</td><td colspan="2">200 cycles</td></tr>
<tr><td>Branch predictor</td><td>Perfect</td><td>Combined
· bimodal and gshare
· 4K meta-table size</td></tr>
<tr><td>L2 MSHRs</td><td>Unlimited</td><td>8</td></tr>
<tr><td>L2 Data prefetcher</td><td>—</td><td>Tagged prefetcher</td></tr>
</table>

Table 6: Baseline and realistic superscalar processor configurations to evaluate `PDCM`.

further evaluated by individually adding a key superscalar processor artifact to the baseline configuration in consideration. Finally, the realistic superscalar processor configuration incorporating all the artifacts is used to evaluate PDCM. The entire SPEC2K benchmarks are used for evaluation.

To demonstrate the efficacy of PDCM, a PDCM-based trace-driven simulator (`PDCM`) is employed and the simulation results are compared with that of `sim-outorder`, a detailed execution-driven simulator. `sim-outorder` has been largely used as a counterpart when verifying a new simulation method or an analytical model for superscalar processors [13, 24, 36, 45, 68, 80]. The main metrics are *CPI error* and *relative CPI change*. CPI error is defined as $(CPI_{pdcm} - CPI_{soo})/CPI_{soo}$, where CPI_{pdcm} is the CPI obtained with `PDCM` and CPI_{soo} is the CPI obtained with `sim-outorder`. CPI error is used to show the percentage of difference in cycle count when it is measured with `PDCM` and `sim-outorder`. The average CPI error is obtained by taking the arithmetic mean of the absolute CPI errors, where the absolute value of the CPI error shows the magnitude of the difference. Relative CPI change is defined as $(CPI_{conf2} - CPI_{conf1})/CPI_{conf1}$, where CPI_{conf1} is the CPI of a base configuration and CPI_{conf2} represents the CPI of a revised configuration [47]. Relative CPI change is used to

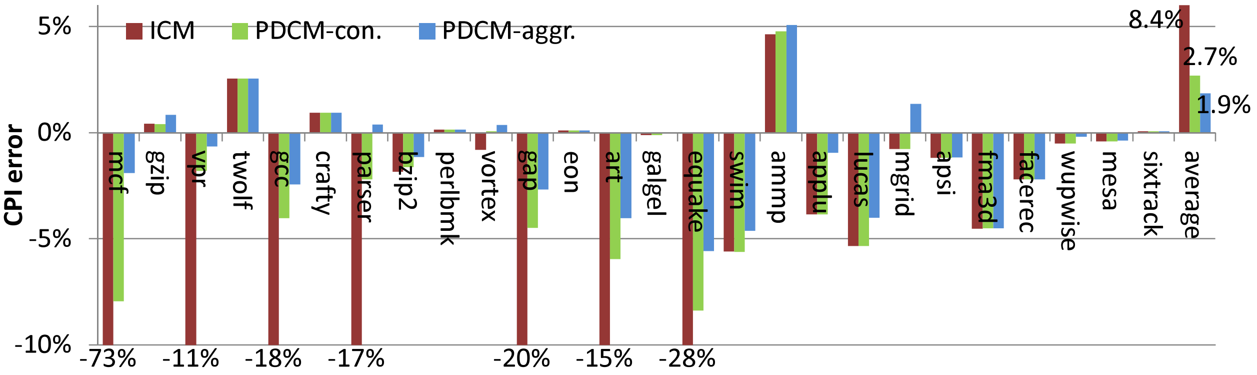


Figure 15: The CPI errors of the entire SPEC2K benchmarks using the baseline configuration in PDCM. "ICM" is the independent cache miss model, and "PDCM-eager" and "PDCM-lazy" stand for eager and lazy estimation of the dependent trace item's processing time.

measure the performance change of the revised configuration relative to the performance of the baseline configuration.

Relative CPI difference is used to compare the performance change amount shown by `sim-outorder` and `PDCM`. Relative CPI difference is defined as $|rel\_cpi\_chg_{soo} - rel\_cpi\_chg_{pdcm}|$, where $rel\_cpi\_chg_{soo}$ and $rel\_cpi\_chg_{pdcm}$ are the relative CPI change shown by `sim-outorder` and `PDCM`, respectively.

Lastly, the capability of `PDCM` to reproduce the behavior of a superscalar processor by tracking the temporal changes in memory access patterns is shown. Program execution is divided into intervals and a histogram is generated to collect the frequency of the distance (in cycles) between two consecutive memory accesses in each interval. The frequency of the collected distances in `PDCM` and `sim-outorder` is compared to examine how closely `PDCM` reproduces the off-core memory access patterns of `sim-outorder`.

3.4.6 Evaluation result

3.4.6.1 Accuracy of PDCM In this section, the evaluation results of PDCM using the baseline and realistic configurations are presented.

• **PDCM with the baseline configuration.**: Figure 15 presents the CPI error of the entire 26 SPEC2K benchmarks with the baseline configuration. The CPI error of the independent cache miss model (ICM) [45]—equivalent to PDCM without taking the data dependency into account—is shown to reveal the importance of honoring the data dependency between trace items. Since ICM does not consider the dependency between trace items, it suffers large CPI errors for the benchmarks that have many dependencies between L1 cache misses, e.g., *mcf* (−73% CPI error). The CPI errors are significantly reduced with PDCM.

The figure presents the results of PDCM with the eager and the lazy approaches (used when estimating the time to process the dependent trace items, see Section 3.4.4.2) separately. The results show that the lazy approach has lower CPI errors in general than the eager approach because there are often intervening instructions dependent on the parent trace item. Accordingly, the lazy approach is used in the remainder of this section. The CPI errors of the SPEC2K benchmarks range from −6% (*equake*) to 5% (*ammp*) with an average of 1.9%.

The results show that some benchmarks, such as *ammp*, show positive CPI error even with ICM. This is because PDCM does not take into account the overlap between the outstanding L2 cache misses and the cycle count recorded in a pending trace item. Consider the case when there is only one trace item in the ROB accessing the main memory and a pending trace item waiting for the trace item to commit. Since the cycle count in the pending trace item is the time spent on executing the instructions between the two trace items, a portion of that cycle should be overlapped with the memory access. However, in this work, the entire cycle count is simply used to estimate the processing time of the pending trace item in trace simulation. Note that the individual direction of the CPI error using a particular configuration is of relatively small interest (as compared with experiments spanning multiple configurations, e.g., Section 3.4.6.3). What is more important at this point is the magnitude, which is fairly small.

The small CPI errors show that PDCM can accurately model the baseline configuration, but PDCM is also robust to the variation in processor's inherent parameters. To study the sensitivity of the model, PDCM is evaluated with different ROB size, L1 data cache size, and issue-width. The experiment results show that the CPI errors were less than 3% when different machine configuration is used in trace generation. Table 7 summarizes the studied results.

Different ROB sizes				
size	32	64	128	256
Avg. CPI error	1.5%	1.9%	2.3%	2.8%

Different L1 data cache sizes			
size	8KB	16KB	64KB
Avg. CPI error	2.5%	2.1%	1.8%

Different issue-width		
width	2	8
Avg. CPI error	2.2%	2.1%

Table 7: The accuracy of PDCM with different processor core configurations. The processor's dispatch-width and commit-width are assumed to be identical to the processor's issue-width.

Finally, although a fixed main memory access latency is assumed in this work, the accuracy of PDCM does not depend on the off-chip access latency. To evaluate PDCM with various main memory access latencies, experiments were conduced with a DRAM model in sim-outorder and PDCM that has 16 banks (8 banks x 2 ranks) with 16KB row size and an open-page policy. In the experiments, the main memory access latency is set to 80 cycles when a page hit occurs in a bank, and 180 cycles when a page miss occurs in a bank. The average CPI error of PDCM was 4.1% over the entire SPEC2K benchmarks, which demonstrates that PDCM is accurate with non-constant off-chip access latency.

- **Effect of instruction caching in PDCM.**: Up to this point, the trace files are generated assuming a perfect instruction cache. To study the effect of instruction caching in PDCM, a realistic 32KB instruction cache is employed during trace generation.

The results show that incorporating the instruction caching artifact in PDCM does not affect the accuracy of PDCM. There were only 7 benchmarks that showed a relative CPI change larger than 0% using sim-outorder after incorporating a realistic instruction cache to the baseline configuration: *gcc* (10%), *crafty* (4%), *parser* (1%), *perl* (13%), *vortex* (2%), *eon* (1%), and *apsi* (2%). The relative CPI difference of the 7 benchmarks was 0.3% on average and the largest relative CPI change was shown by *perl* from both sim-outorder and PDCM. The average CPI error of the entire 26 SPEC2K benchmarks was 1.8% on average.

- **Effect of branch prediction in PDCM.**: The trace files used so far are generated

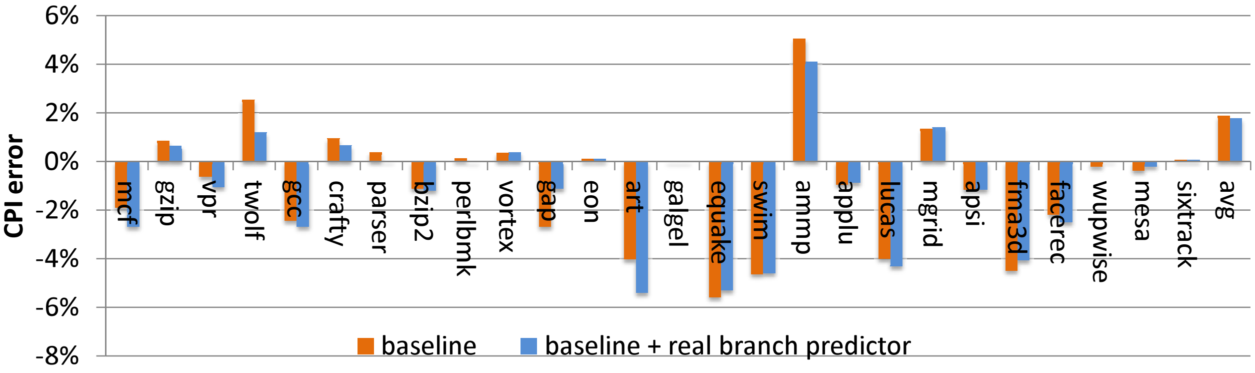


Figure 16: The CPI errors before (*base*) and after (*base + real branch predictor*) incorporating a realistic branch predictor in PDCM.

with a perfect branch predictor. Let us now study how the branch mis-predictions in trace generation can affect trace simulation accuracy. PDCM is driven by trace items generated with a combined branch predictor (bimodal and gshare) in trace generation. Note that `sim-outorder` was configured with the identical branch predictor.

Figure 16 compares the CPI errors before and after incorporating a realistic branch predictor to the baseline configuration. The results show that incorporating the branch prediction artifact in PDCM does not affect the accuracy of PDCM. The largest branch misprediction penalties was shown by *perl* from both `sim-outorder` and PDCM. The relative CPI change of *perl* after employing a realistic branch predictor was 48% in both `sim-outorder` and PDCM. The relative CPI difference of the entire SPEC2K benchmarks was 1% on average.

One might question the validity of the timing information recorded in the trace items. In trace generation, since a perfect L2 cache is used, a fixed L2 cache hit latency is returned on each L1 miss. On the other hand, different latencies are returned depending on the result of the L2 cache access in `sim-outorder`. If the memory access latency significantly affects the branch prediction accuracy, it is difficult to correctly model the branch mis-prediction penalties using the cycle counts in the trace items. To investigate this aspect, the effect of the memory access latency on branch prediction accuracy is quantified using the eight representative benchmarks in the SPEC2K benchmark suite [34], as shown in Table 8.

The branch prediction accuracy was collected separately from two simulation runs. The

	% of stable branch instructions in the program			
stability	≤ 0.005	≤ 0.01	≤ 0.02	≤ 0.03
mcf	84%	91%	94%	95%
gcc	79%	84%	88%	91%
gzip	84%	88%	91%	95%
twolf	82%	89%	93%	96%
fma3d	96%	97%	97%	97%
applu	91%	92%	93%	94%
mesa	87%	88%	88%	88%
equake	88%	91%	94%	95%

Table 8: The percentage of stable branch instructions in the benchmarks.

first simulation uses a fixed L2 hit latency and the second simulation uses a random memory access latency—any integer number between the L2 hit latency (12) and a long memory access latency (400)—on L1 misses. Using `sim-outorder`, for a given branch instruction (say A), the number of correct branch predictions was collected from A with a fixed latency ($n\_correct_{\mathsf{A}-fixedlat}$) and a random latency ($n\_correct_{\mathsf{A}-randomlat}$), and the number of total branch predictions made from A (n_total_{A}). The *stability* of a given branch instruction A is defined by:

$$stability = \frac{|n\_correct_{\mathsf{A}-fixedlat} - n\_correct_{\mathsf{A}-randomlat}|}{n\_total_{\mathsf{A}}}$$

For instance, the last column in Table 8 shows that 88% of *gcc*'s branch instructions and 97% of *fma3d*'s branch instructions are stable, when the threshold is 0.03. The high stability values in the table suggest that the L1 cache miss latency does not significantly change the branch prediction accuracy.

In the following, `PDCM` is evaluated using the relative CPI change metric with the benchmarks that show a change in CPI when superscalar processor artifacts are added in the baseline configuration.

• **Effect of L2 data prefetching in PDCM.**: Figure 17 compares the relative CPI change reported by `sim-outorder` and `PDCM`, when a tagged L2 data prefetcher is added in the baseline configuration. The results show that `PDCM` can accurately model the effect of L2 data prefetching. The two largest beneficiaries were *swim* and *mgrid* as shown by both

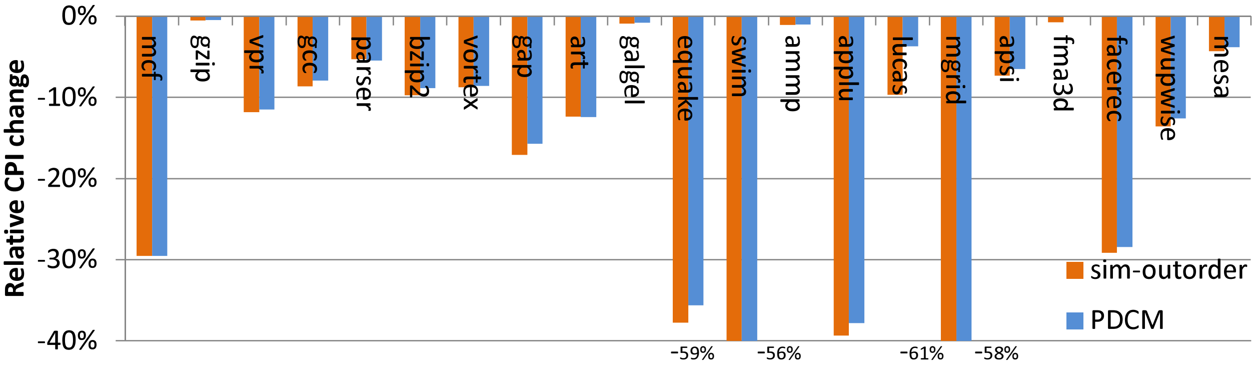


Figure 17: The relative CPI changes when a tagged L2 data prefetcher is incorporated in the baseline configuration in `PDCM`.

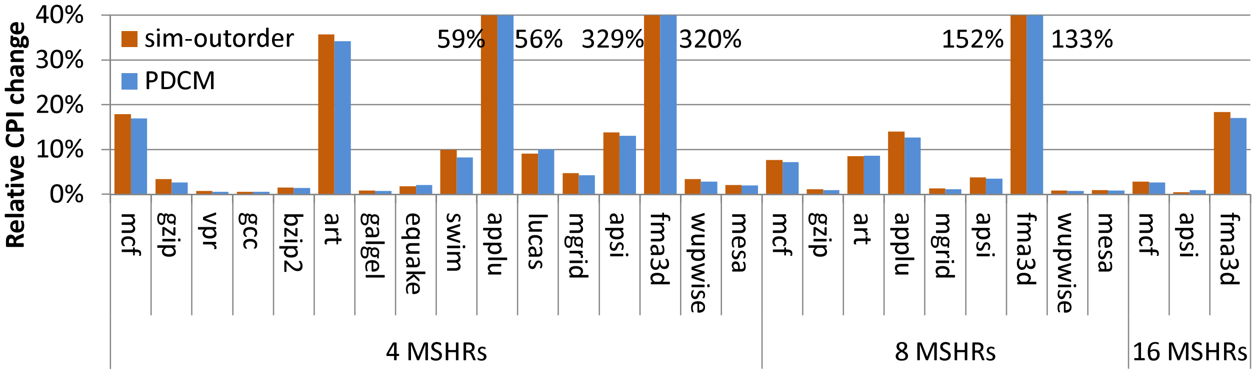


Figure 18: The relative CPI changes when 4, 8, and 16 MSHRs are used, compared with unlimited MSHRs, in `PDCM`. Among the entire 26 SPEC2K benchmarks, only the results of the benchmarks that showed at least 1% relative CPI change from either `sim-outorder` or `PDCM` are presented.

`sim-outorder` and `PDCM`. The relative CPI difference of the benchmarks in the figure was 1% on average. The CPI error with the tagged prefetcher in the baseline configuration was 1.6% on average.

• **Effect of limited L2 MSHRs in PDCM.**: Figure 18 compares the relative CPI change obtained with `sim-outorder` and `PDCM`, when limited number of L2 MSHRs is applied to the baseline configuration. Since the number of outstanding L2 cache misses is limited by the number of L2 MSHRs, the CPI increases with fewer L2 MSHRs.

The results show that `PDCM` can closely follow the relative CPI change of `sim-outorder`.

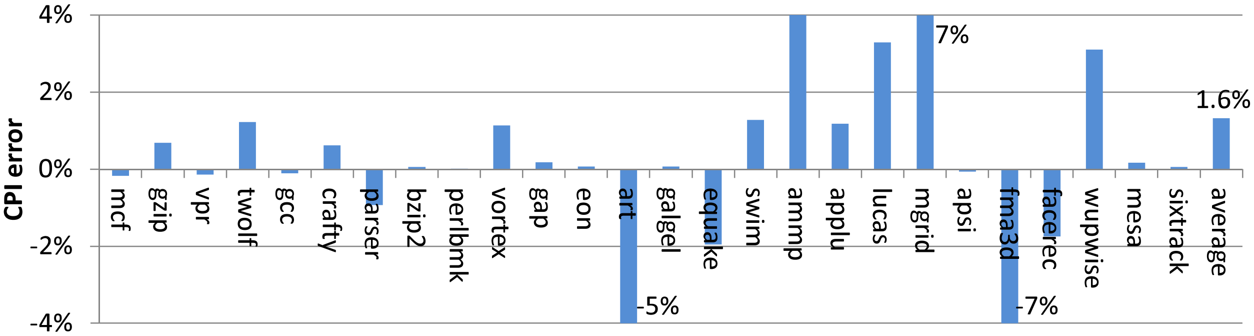


Figure 19: The CPI errors of the SPEC2K benchmarks using the realistic configuration in PDCM.

The relative CPI difference of the benchmarks in the figure was 2% on average. *fma3d* is particularly sensitive to the number of MSHRs because it has a very high L2 cache miss rate and 72% of the execution time comes from L2 cache miss penalties. Moreover, L2 cache accesses occur in a very close distance only in certain periods during program execution. PDCM was able to reproduce this unique behavior of *fma3d*. The average CPI error was 2.1%, 2.2%, and 1.9%, when sim-outorder and PDCM both used 4, 8, and 16 MSHRs respectively.

• **PDCM with the realistic configuration.**: Finally, the simulation results using the realistic superscalar processor configuration with PDCM is presented. The trace files used in this section are generated with branch mis-predictions and instruction cache misses, and sim-outorder was also modified to model the realistic superscalar processor. The CPI errors of the entire 26 SPEC2K benchmarks are reported in Figure 19. The increased CPI error of *mgrid*, compared with 1% CPI error with the baseline configuration, comes from the error when modeling the data prefetching effect. The results show that PDCM achieves a very high accuracy with an average error of 1.6%, which is even smaller than the average error (1.9%) with the baseline configuration.

Since CPI error is a metric that is averaged over the entire simulation, it does not show how accurately PDCM is modeling the superscalar processor performance over program execution. To further examine the accuracy of PDCM, PDCM is evaluated with a series of CPI errors measured over the program execution. The program execution is divided by an interval of 1M instructions. Then in each interval, CPIs are measured using sim-outorder

Benchmark	Avg.	Min.	Max.	Benchmark	Avg.	Min.	Max.
mcf	1.3%	0.0%	6.5%	*art*	4.6%	0.0%	11.3%
gzip	6.6%	3.2%	13.3%	*galgel*	0.6%	0.0%	13.5%
vpr	5.0%	0.6%	11.4%	*equake*	2.9%	0.0%	18.3%
twolf	0.6%	0.0%	1.1%	*swim*	1.6%	0.1%	7.5%
gcc	1.8%	0.0%	24.9%	*ammp*	4.1%	0.0%	10.8%
crafty	3.6%	2.4%	15.8%	*applu*	3.3%	0.1%	12.7%
parser	3.4%	0.0%	21.9%	*lucas*	5.6%	0.0%	27.4%
bzip2	2.9%	0.1%	20.0%	*mgrid*	5.8%	0.0%	12.2%
perl	6.3%	4.6%	8.4%	*apsi*	0.6%	0.0%	23.7%
vortex	2.8%	1.0%	14.2%	*fma3d*	3.5%	0.6%	18.3%
gap	3.5%	2.0%	12.0%	*facerec*	2.0%	0.6%	3.3%
eon	4.2%	2.0%	8.6%	*wupwise*	2.8%	0.2%	9.1%
				mesa	1.7%	0.2%	21.9%
				sixtrack	7.4%	7.1%	8.3%

Table 9: The average, minimum, and maximum CPI errors of `PDCM` observed throughout a program execution using the realistic configuration.

and `PDCM` to compute the CPI error of `PDCM`.

Table 9 shows the average, minimum, and maximum CPI errors of `PDCM` that were observed from 1,000 intervals using the entire SPEC2K benchmarks. The results show that there are benchmarks that show a large CPI error at some point during program execution. However, overall, both `PDCM` and `sim-outorder` showed a very similar trend in CPI change. Figure 20 shows an example with *lucas*, which has the largest CPI error in an interval (27.4%) among all 26 SPEC2K benchmarks. The figure presents `sim-outorder` and `PDCM` showing a similar trend in CPIs measured over 1,000 intervals. `PDCM` showed higher than 20% CPI error when simulating in intervals between interval #104 and #110 (the region where the first spike appears).

3.4.6.2 Reproducing temporal uncore access behavior In the above experiments, the CPI error and relative CPI change computed over the entire execution span were used as the main metrics to evaluate how closely PDCM approximates a realistic superscalar processor's performance. In what follows, this work focuses on two aspects of `PDCM` that are

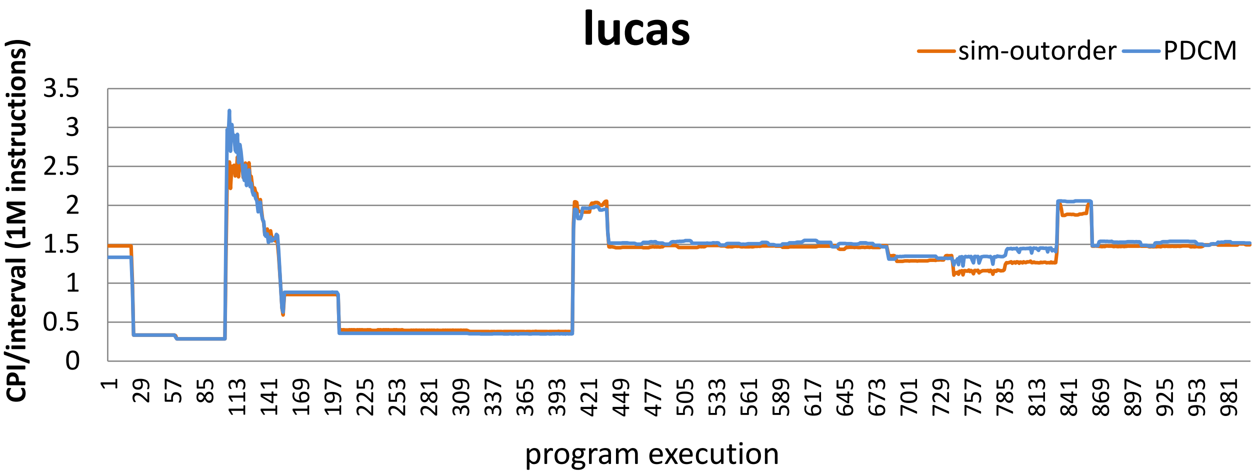


Figure 20: The change in CPI of *lucas* shown by `sim-outorder` and `PDCM` while simulating 1B instructions (1,000 intervals).

relevant at the system level: (1) Does PDCM changes how a processor core exercises "uncore" resources such as L2 cache and memory controller? and (2) Can PDCM predict a program's relative performance when uncore parameters such as L2 cache size are changed? These aspects are especially important when evaluating a workload on a multicore architecture where uncore resources are subject to contentions.

To explore the first aspect, for each benchmark, histograms of the distance (in cycles) between two consecutive off-chip accesses (from L2 cache misses, writebacks from the L2 cache, or L2 data prefetching) are built in each interval of 100M instructions from both `sim-outorder` and `PDCM`. Each bin in a histogram represents a specific range of distances, and the value in a bin represents the frequency of distances that fall into the specified range.

In this experiment, `PDCM` is assumed to preserve the temporal off-chip access patterns of `sim-outorder` if the frequency of distances between two consecutive off-chip accesses that occur in each interval is similar to `sim-outorder`. The histogram is generated for 10 consecutive intervals from both `sim-outorder` and `PDCM`. The following metric

$$Similarity = \frac{\sum_{i=0}^{n} \mathrm{MIN}(bin\_soo_i, bin\_pdcm_i)}{\sum_{i=0}^{n} bin\_soo_i}$$

is used to compare `sim-outorder` and `PDCM` with a single number, where i is the bin index and n is the total number of bins. bin_soo_i and bin_pdcm_i are the frequency value in i_{th} bin collected by `sim-outorder` and `PDCM`, respectively. High similarity implies `PDCM`'s ability to preserve the off-chip access pattern of `sim-outorder`. If the similarity is 1, it suggests that the observations made by the two simulators are identical.

Figure 21 depicts the representative interval of *mesa* and *parser*. Only the representative intervals of the benchmarks that show the highest (*mesa*) and lowest (*parser*) similarity are shown for clear presentation. *mesa* shows that `sim-outorder` and `PDCM` agree well on the off-chip access behavior, while *parser* shows that `sim-outorder` and `PDCM` disagree somewhat on the frequency of the distances between isolated memory accesses.

Table 10 presents the computed average *Similarity* over all intervals for all SPEC2K benchmarks. Note that the similarity values of some benchmarks were affected by having a few memory accesses in an interval of one simulator, while the other simulator not showing any memory accesses in the same interval. For instance, *fma3d* showed two memory accesses in the first interval with `PDCM`, while having no memory access in the first interval with `sim-outorder`. If the first two memory accesses with `PDCM` were not taken into consideration, the *Similarity* of *fma3d* increases from 85% to 99%. Overall, `PDCM` preserves the memory access behavior of `sim-outorder` closely. Most benchmarks, 19 out of 26, showed 90% or higher similarity.

3.4.6.3 Predicting the performance with different uncore parameters I now attempt to answer the question of "Can PDCM correctly predict the performance of a new machine configuration given the performance of a baseline configuration?" The ability to predict relative performance (i.e., performance trend) is often more important in a system performance study. In the following experiment, the realistic superscalar processor configuration is used as the reference point and five new configurations that differ in one of their L2 cache or main memory parameters (described in Table 11) are simulated. Note that `PDCM` used the same traces produced to study the realistic superscalar processor configuration in Table 6 for all five different configurations.

The results in Table 11 show that `PDCM` was able to project the relative performance very closely to sim-outorder. First of all, the performance change direction (positive or negative), was predicted correctly 100% of the time. Furthermore, Table 11 shows that the relative CPI change seen by each benchmark and each configuration, was nearly identical between the two simulators for most of the benchmarks. *gcc* showed a large relative CPI difference when L2 cache size was reduced (Conf. 1). *gcc*'s CPI increased 78% in sim-outorder when L2 cache size is reduced from 2MB to 1MB, whereas PDCM increased CPI by only 16%. PDCM shows smaller CPI increase because it has smaller L2 cache misses with 1MB L2 cache than sim-outorder. The different number of L2 cache misses is caused by not generating trace items for L1 instruction cache misses. As mentioned in Section 3.4.4.4, instead of generating extra trace items for instruction cache misses, the effect of instruction caching is modeled by employing a realistic instruction cache in trace generation and recording the increased cycle count in trace items. However, such simple approach does not accurately capture the case when L1 instruction cache misses increase the number of L2 cache misses with smaller L2 cache. In general, fairly accurate projections of the relative performance were obtained from PDCM. The average relative CPI difference of the five configurations ranged from 0.3% (larger L2 cache) to 3.4% (smaller L2 cache).

In summary, the results presented in Figure 21, Table 10, and Table 11 suggest that PDCM is amenable for use in a multicore simulation environment [46, 42]. 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. The techniques can be applied to multithreaded shared memory applications if individual threads can be traced [42]. One can reliably study the overall system behavior thanks to the capability of the presented technique 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.

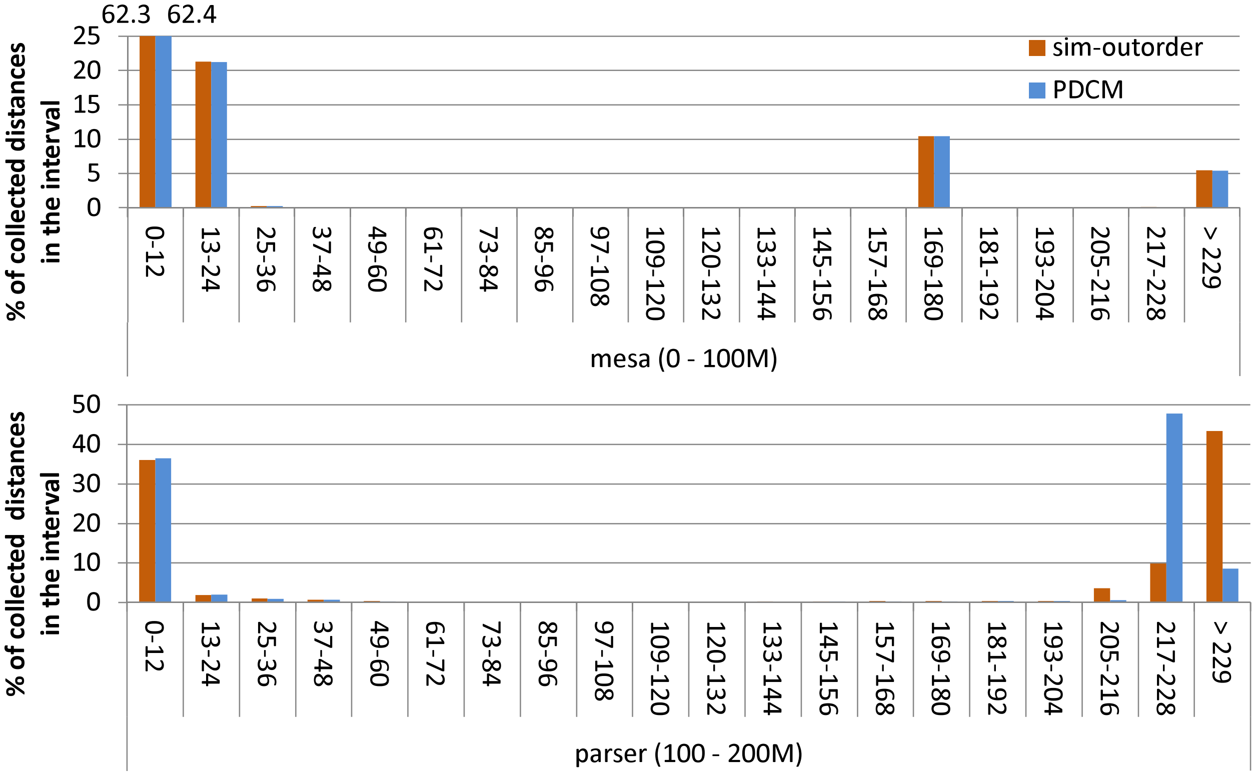


Figure 21: The histogram of collected distances in an interval is depicted to compare the distance (in cycles) between two consecutive L2 cache misses in `sim-outorder` and `PDCM`. The x-axis represents the bins used to generate the histogram and the y-axis represents the percent of collected distances in the interval of 100M instructions. The bin size is 12 cycles.

Similarity	Benchmark (similarity)
< 90%	*parser* (81%), *gzip* (83%), *fma3d* (85%) *mgrid* (86%), *facerec* (88%), *swim*, *eon* (89%)
≥ 90%	*art*, *gap* (90%), *galgel*, *gcc* (91%) *ammp*, *applu*, *equake*, *mcf* (92%), *vpr* (93%)
	twolf (94%), *lucas* (95%), *wupwise* (96%)
	apsi, *bzip2*, *crafty* (97%) *perl*, *vortex* (98%), *mesa*, *sixtrack* (99%)

Table 10: The similarity in memory access patterns between `sim-outorder` and `PDCM` (shown in percentage).

Benchmark	Conf. 1	Conf. 2	Conf. 3	Conf. 4	Conf. 5
mcf	1%	0%	1%	1%	0%
gzip	0%	0%	0%	0%	1%
vpr	1%	0%	0%	0%	1%
twolf	12%	0%	0%	0%	11%
gcc	62%	0%	0%	1%	6%
crafty	0%	0%	0%	0%	1%
parser	0%	1%	0%	0%	3%
bzip2	1%	0%	0%	0%	3%
perl	0%	0%	0%	0%	8%
vortex	0%	0%	0%	0%	1%
gap	0%	0%	0%	0%	1%
eon	0%	0%	0%	0%	1%
art	3%	3%	2%	2%	1%
galgel	3%	0%	0%	0%	2%
equake	0%	0%	0%	0%	0%
swim	0%	0%	3%	1%	0%
ammp	3%	0%	0%	0%	1%
applu	0%	0%	1%	1%	0%
lucas	0%	0%	0%	0%	1%
mgrid	0%	0%	1%	4%	0%
apsi	0%	0%	0%	0%	2%
fma3d	0%	0%	1%	1%	0%
facerec	0%	1%	0%	0%	1%
wupwise	0%	0%	0%	0%	1%
mesa	0%	0%	0%	0%	0%
sixtrack	1%	0%	0%	0%	0%
Avg. Error	3.4%	0.3%	0.4%	0.5%	1.8%

Table 11: The relative CPI differences between `sim-outorder` and `PDCM`. The five configurations are identical to the realistic configuration (Table 6) except a single parameter. In Configuration 1 (Conf. 1) and 2 (Conf. 2), the L2 cache is 1MB and 4MB instead of 2MB ("smaller L2 cache" and "larger L2 cache"). In Configuration 3 (Conf. 3) and 4 (Conf. 4), the memory latency is 100 cycles and 300 cycles instead of 200 cycles ("faster memory" and "slower memory"). In Configuration 5 (Conf. 5), the L2 hit latency is 20 cycles instead of 12 cycles ("slower L2 cache"). The performance change directions observed from the two simulators were identical.

3.4.6.4 Simulation speed and storage requirements of PDCM Lastly, the simulation speed and storage requirements of `PDCM` and a full trace-driven simulation strategy, a widely practiced simulation method [4] is presented. The trade-off between the two methods is clear. The biggest advantage of using `PDCM` over the full trace-driven simulation is its fast simulation speed. On the other hand, the full trace-driven simulation strategy has the advantage of being more accurate with the complete information of all instructions executed.

The simulation speeds of the two trace simulation methods are compared using the speedups achieved over `sim-outorder` with the baseline configuration. Because the full detail of the target superscalar processor operation per every supported instruction is modeled, the implementation of the full trace simulator is essentially identical to `sim-outorder`. The observed simulation speedups of `PDCM` range from 3.8× (*gcc*) up to 582.58× (*eon*) and their average (geometric mean) was 62.5×. Note that the trace generation time was not included when measuring the speedup. The trace generation was 1.24x slower than an execution-driven simulation on average (geometric mean) over the entire SPEC2K benchmarks. On the other hand, the speedups with the full trace-driven simulation was limited, ranging from 1.06× (*mcf*) to 1.35× (*sixtrack*). The average speedup was only 1.18×.

The observed absolute trace simulation speeds with `PDCM` using the realistic configuration range from 2.3 MIPS (*gcc*) to 428.3 MIPS (sixtrack) and their average was 48.3 MIPS (geometric mean), as shown in Figure 22. The simulation speed of fast cycle-accurate detailed execution-driven simulators are about 0.5 MIPS on a 2GHz Pentium 4 [80].

The simulation speedups achieved with `PDCM` over `sim-outorder` using the realistic configuration range from 3.5× (*gcc*) and 406.5× (*eon*). The average (geometric mean) simulation speedup was 55.3× on average. `PDCM`'s absolute trace generation speed, using the realistic configuration, ranges from 443 KIPS (*gap*) to 1171 KIPS (*lucas*) and their average was 756 KIPS (geometric mean).

In the current implementation, a single trace item in a full trace is 12B. Because trace items are generated for 1.1B instructions, each trace file is 12.3GB. A single trace item with `PDCM` is 20B, and the average trace file size is 1.5GB, ranging from 20MB (*eon*) to 9.6GB (*gcc*). 22 out of 26 benchmarks were less than 1.9GB, and 14 out of 26 were less than 1GB. Certain benchmarks require many trace items because of delayed hits, especially *gcc*. Since

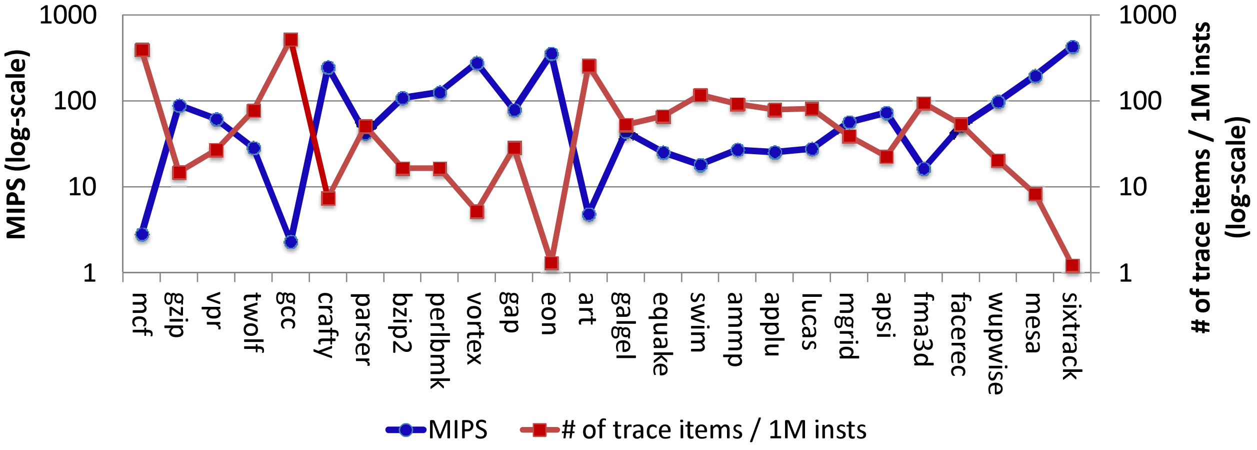


Figure 22: The relationship between the simulation speed and trace file size in PDCM.

not all delayed hits may be needed for accuracy, the trace file size can be reduced significantly if unnecessary delayed hits are filtered during trace generation as a possible optimization. Trace file sizes were slightly larger with the realistic configuration with an average of 1.5GB, and range from 25MB (*eon*) to 9.7GB (*gcc*).

As expected, simulation results of the full trace-driven simulation method were almost identical to `sim-outorder` with 0% CPI error on average. PDCM is not as accurate, but the error is very limited as discussed in this section. Considering the fast simulation speed, small error, and much smaller storage overheads, PDCM is a more attractive simulation method than the full trace-driven simulation, especially in the early design stages.

3.5 SUMMARY

The three cache miss models examined in this section have different strengths and weaknesses. The isolated cache miss model works well when the simulated program has a high L1 cache hit rate and isolated L1 cache misses. 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 simulator would determine the timing of each trace item in the trace using the ROB status constructed during trace simulation. However, if there are dependencies among trace items, this overly optimistic model becomes inaccurate.

The third model, the pairwise dependent cache miss model (PDCM), 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, PDCM 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 15 showed that PDCM outperforms the independent cache miss model in terms of the CPI error metric. I conclude that PDCM is the most accurate model among the three when modeling the superscalar processor performance from reduced traces. Compared with a detailed execution-driven simulation method, PDCM achieves an absolute simulation speed of 48 MIPS on average (geometric mean) while giving sufficiently small errors across benchmarks (less than 3% on average). PDCM also robustly predicts the relative performance change for different machine configurations. The performance change direction is always predicted correctly and the performance change amount is predicted with small errors of less than 4% on average.

4.0 TRACE SIMULATION WITH ABSTRACT TIMING INFORMATION

The pairwise dependent cache miss model (PDCM) provides accurate simulation results using timing-aware filtered traces. However, it requires a cycle-accurate timing simulator that models the microarchitecture of a target processor to generate the timing-aware filtered traces. This is a drawback, if a cycle-accurate timing simulator is not available, especially in the early processor design stages. On the other hand, a functional simulator is usually prepared in the early design stages for software development. In-N-Out employs a functional simulator to build an abstract timing model to quickly generate reduced in-order traces, rather than using a detailed timing simulator, and hence, it is more appealing than PDCM in the early design stages.

4.1 OVERVIEW

The overall structure of In-N-Out is illustrated in Figure 23. In-N-Out uses an *abstract timing model* to quickly generate *reduced traces (L1 filtered traces)* on L1 data cache misses and writebacks. Since the abstract timing model is based on a functional simulator, unlike PDCM, the traces are generated in program order. In this dissertation, the `sim-cache` [2] simulator is modified to implement an abstract timing model. The abstract timing information is gathered by monitoring the dependencies between instructions. The monitored dependency are the data dependency, memory dependency, and the dependency created by the limited architectural resources or processor artifacts.

The reduced trace is fed into the trace simulator with the target machine definition. The *target machine definition* includes the processor's ROB size and the configuration of the

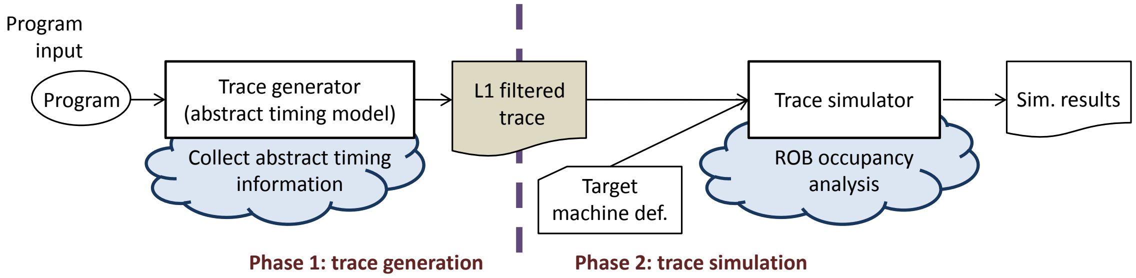


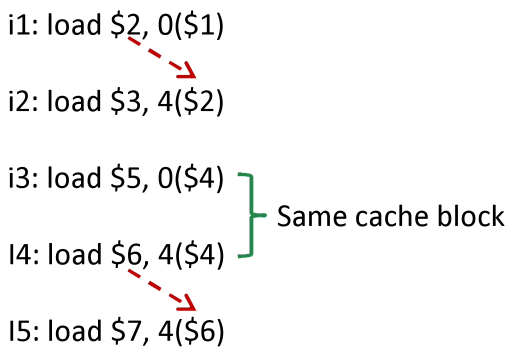
Figure 23: Overall structure of In-N-Out. It uses an abstract timing model to generate reduced traces (L1 filtered traces).

uncore components. The *trace simulator* runs the out-of-order trace simulation algorithm by dynamically reconstructing the ROB state and exploiting the recorded abstract timing information in trace items. Finally, the simulation results are obtained from the trace simulator. Compared with the trace-driven simulation models introduced in Chapter 3, the filtered trace generation is simpler and faster because an abstract timing model is used.

The following sections present how the performance of superscalar processors is modeled using reduced in-order traces. First, the trace generation approach is discussed, followed by the details of the key design issues and the trace simulation algorithm. The quantitative evaluation results are reported at the end of this section.

4.2 TRACE GENERATION IN IN-N-OUT

In Section 3, it was shown that superscalar processor performance can be accurately modeled using timing-aware traces generated from a cycle-accurate timing simulator. However, in In-N-Out framework, since the trace generator is based on a functional simulator, accurate timing information of the processor core cannot be collected. The abstract timing information can be collected considering the dependencies between instructions and the key architectural parameters, such as the ROB size and the processor's dispatch width. To collect the abstract timing information, the following three types of dependencies are monitored



```
i1: load $2, 0($1)
i2: load $3, 4($2)
i3: load $5, 0($4)
I4: load $6, 4($4)
I5: load $7, 4($6)
```

Figure 24: An example of instruction data dependency in a program.

during trace generation: (1) data dependency, (2) memory dependency, and (3) microarchitectural dependency.

4.2.1 Data dependency in superscalar processor

If an instruction depends on the data created by a preceding instruction, there exists a data dependency between the two instructions. In Figure 24, load instructions i2 and i5 depend on the data fetched by the load instructions i1 and i4, respectively, to compute their memory address.

Besides the explicit data dependency between trace items, an implicit data dependency may exist via a "delayed hit" as described in Section 3.4.2. A delayed hit occurs when a memory instruction accesses a cache block that is still in transit from the lower-level cache or the main memory. The second access to the block after a miss is registered as a hit, but the access has to wait for the data to be brought from the memory. Consider an L1 data cache miss that depends on an L1 delayed hit. This L1 data cache miss must be processed after the previous data cache miss that caused the delayed hit, since the delayed hit has to wait until the data is brought by the previous data cache miss [13]. An example is shown in Figure 24. In the example, load instructions i3 and i4 access the same cache block. Assume instruction i3 is issued before i4 and misses in both L1 and L2 caches. Instruction i5 does not depend on i3, but since i3 and i4 access the same cache block and i5 has to wait until i4 returns from the main memory, there exists an implicit data dependency between i5 and i3 via i4.

In the PDCM framework (see Section 3.4.2), to expose all data dependencies between trace items during the trace simulation phase, the trace items are generated for L1 hits that may become L1 delayed hits during trace simulation. However, certain benchmarks, such as *gcc*, show a significantly large increase in trace file size when delayed hit trace items are generated as discussed in Section 3.4.6.4. Moreover, since the trace simulation speed is determined by the number of simulated trace items, it is important to minimize the number of trace items in a trace file.

In In-N-Out, since not all delayed hit trace items are necessary for accurate simulation, unnecessary delayed hit trace items are filtered. During trace generation, when an L1 data cache miss occurs, the cache block is labeled with the instruction sequence number (ISN) of the memory instruction that generated the miss. Later, when a memory instruction accesses the same cache block, the labeled ISN on the cache block is used to notify the memory instruction that there is a trace item it indirectly depends on. Note that a trace item generated by a load instruction can be marked as a dependent of a trace item generated by a store instruction, if it depends on a delayed hit created by the store instruction. Section 4.3.1 describes in detail how the delayed hit trace items are treated during trace simulation.

The data dependency between instructions are identified based on the registers accessed by the instructions. When an instruction is processed, ISN is given to the instruction in program order. When an instruction writes to a register, it labels the output register with its *ISN*. Later, when a different instruction reads data from the same register, the labeled *ISN* is used to identify the existing dependency. For example, in Figure 25, instruction i2 writes its *ISN* in its output register and then the dependency between i2 and i3 is detected when i3 reads from i2's output register. This creates a dependency chain (DC_2) originating from i2. The length[1] of the dependency chain is incremented when a new instruction is included in the dependency chain. When two separate dependency chains merge at one instruction (e.g., instruction i4 in Figure 25), the length of the longer dependency chain is used.

[1] The length of a dependency chain is defined as the sum of the execution latencies of the instructions on the dependency chain.

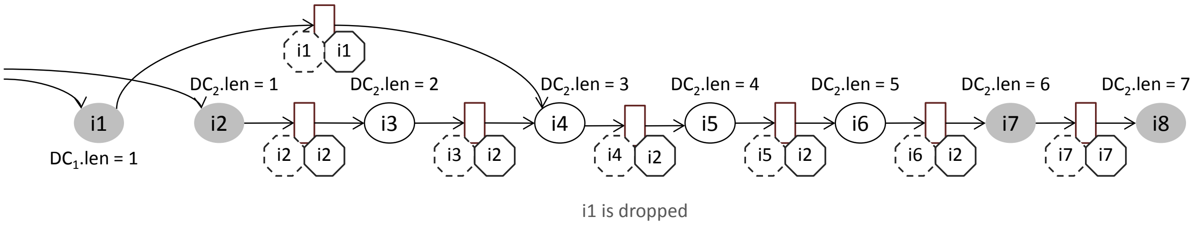
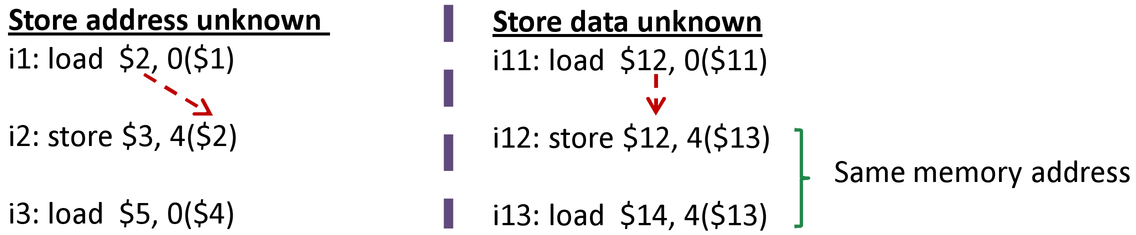


Figure 25: An example of eight instructions constituting two dependency chains. Each square represents a register and each circle represents an instruction. Circle filled in darkish color represents an instruction that generates a trace item. The dashed octagon on the square indicates the last instruction that produced the data and the solid octagon indicates the last trace item in the dependency chain.

The dependency between trace items is also identified and recorded in the dependent trace item. The recorded dependency information includes the ISN of the parent trace item and the distance (in terms of the instruction execution latency) between the two trace items. Figure 25 explains how the dependency between trace items is detected. When a trace item is generated, the ISN of the trace item is propagated in the dependency chain by its descendants. For instance, instruction i2 and i7 are trace items in the example. The ISN of i2 is passed on by the instructions in DC_2 and the dependency between i2 and i7 is detected from the ISNs labeled in i7's input register. For instructions depending on multiple trace items, the most largest ISN in the dependency chain is kept. While storing more than one trace item may improve accuracy, experiment results revealed that storing single ancestor is sufficient.

4.2.2 Memory dependency in superscalar processor

To ensure the correctness of the data read and written to the caches and the main memory, superscalar processor issues the memory instructions considering the memory dependency. During trace generation, if there is a memory dependency between trace items, the memory dependency information is marked in the dependent trace item. Then during trace simulation, the dependent trace item is processed after the memory dependency is resolved. There are two conditions that can create a memory dependency between two memory instructions.



```
i1: load  $2, 0($1)

i2: store $3, 4($2)

i3: load  $5, 0($4)
```

```
i11: load  $12, 0($11)

i12: store $12, 4($13)

i13: load  $14, 4($13)
```

Figure 26: An example of a memory dependency in a program.

One condition is when there is an earlier store instruction with an unknown address. In Figure 26, instruction i2 depends on instruction i1 to compute its memory address. Suppose i1 misses in L1 and L2 caches, and the computed cache block address accessed by i2 and i3 are the same. Instruction i3 does not depend on instructions i1 and i2. However, the processor cannot issue i3 before i1 finishes accessing the main memory because otherwise i3 will fetch stale data from the cache block. Hence, when scheduling load instructions, if there is an earlier store instruction with an unknown memory address, the processor stops scheduling the load instructions behind the store instruction. During trace generation, if a store instruction B depends on a previous trace item A to compute its memory address, the ISN of the trace item A is propagated to all trace items behind the store instruction B. The propagation ends when the ISN of the currently simulated instruction is ROB size number of instructions away from trace item A, or when another memory dependency is identified.

The other condition that incurs memory dependency is when there is an earlier store instruction with an unknown store operand that accesses the same memory address as a later load instruction. In Figure 26, instruction i12 depends on i11 to get its store operand and instructions i12 and i13 access the same memory address. Suppose instruction i12 already knows its store address, but it is waiting for instruction i11 to fetch its store operand from the main memory. Then instruction i13 cannot be issued until i12's store operand is known, because otherwise, i13 will read incorrect data from the cache. During trace generation, if a store instruction B depends on a previous trace item A for its store data, the ISN of trace item A and the memory address of store instruction B are kept. Later, if there is a load instruction C accessing the same memory address as store instruction B, the model determines that there exists a memory dependency between load instruction C and trace item A. The ISN of the trace item A is then propagated to all the instructions that depend on load instruction C. If one of the dependent instructions is a trace item, it will receive the ISN of trace item A.

4.2.3 Microarchitectural dependency in superscalar processor

The microarchitectural dependency exists because of the finite resources in a superscalar processor [25, 26]. The sources of the microarchitectural dependencies are described below.

- **ROB size.** Since an instruction cannot enter the ROB if ROB is full, there exists a dependency between instructions i_N and $i_{N-ROBsize}$. Instruction i_N can enter the ROB only if $i_{N-ROBsize}$ is committed and removed from the ROB.
- **Processor dispatch-width and commit-width.** The number of instructions entering and exiting the ROB per cycle is limited by the processor's dispatch-width and commit-width. For instance, if the processor can dispatch and commit N instructions per cycle, there is a dependency between instruction i and $i + N$. The processor dispatches instruction $i + N$ to the ROB one cycle after instruction i, and the processor commits instruction $i + N$ one cycle after instruction i.
- **Movement between pipeline stages.** There exists at least one clock cycle delay when an instruction moves between pipeline stages. For instance, assume there is an instruction i moving from the dispatch stage to the execute stage. If instruction i enters the dispatch stage in clock cycle N, it enters the execute stage at cycle $N+1$, if there is no dependency to resolve. If instruction i depends on a preceding instruction, i enters the execute stage at cycle MAX($N+1$, $dependency - resolve - time$), where $dependency - resolve - time$ is the clock cycles when the dependency is resolved.
- **Instruction cache miss.** If an instruction cache miss occurs when fetching instruction i at clock cycle N, instruction i enters the dispatch stage at cycle $N + lat$, where lat is the L2 cache hit latency or the main memory access latency.

- **Branch misprediction.** If instruction `i` is a branch instruction and a branch misprediction occurs by instruction `i` at clock cycle N, the instructions in the correct execution path are not fetched until cycle $N + lat$, where lat is the latency required to resolve the branch misprediction. If a branch instruction depends on an L2 cache miss and incurs a branch misprediction, it has to wait until the depending L2 cache miss fetches the data from the main memory. Hence, if a branch misprediction depends on a preceding trace item, the trace items that are generated after the branch misprediction cannot be processed before the preceding trace item is resolved.

4.2.4 Dependence-graph model

To collect the dependency information during trace generation, a dependence-graph model [25, 26] is constructed as shown in Figure 27. Each simulated instruction is represented by three nodes: dispatch node (`D`), execute node (`E`), and commit node (`C`). The three nodes represent the lifetime of an instruction from the time it enters the ROB until it is removed from the ROB. The `D` node represents the dispatch pipeline stage, where instructions are dispatched to the ROB. The `E` node represents the execute pipeline stage, where instructions with no unresolved dependencies are executed. The `C` node represents the commit pipeline stage, where completed instructions are committed and removed from the ROB. The edge in the graph shows the dependency relationship between the nodes. Each edge is weighted according to the latency required to move from one node to the other node.

Table 12 describes the edges in the dependence-graph model. The edges $D_{i-1}D_i$ and $C_{i-1}C_i$, where i is ISN of an instruction, represent the dependencies between two consecutive instructions $i-1$ and i created by the limited number of instructions a processor can dispatch and commit per cycle. The weight of the $D_{i-1}D_i$ ($C_{i-1}C_i$) edge can either be 0, when two consecutive instructions are dispatched (committed) in the same clock cycle, or 1, when the two instructions are separated by the processor's dispatch-width (commit-width). Additional latency may apply to the $D_{i-1}D_i$ edge when an instruction cache miss occurs by instruction $i - 1$. The weight of D_iE_i edge is one clock cycle because moving from a pipeline stage to the next pipeline stage requires at least one clock cycle. The weight of the $\mathsf{E}_i\mathsf{C}_i$ edge is

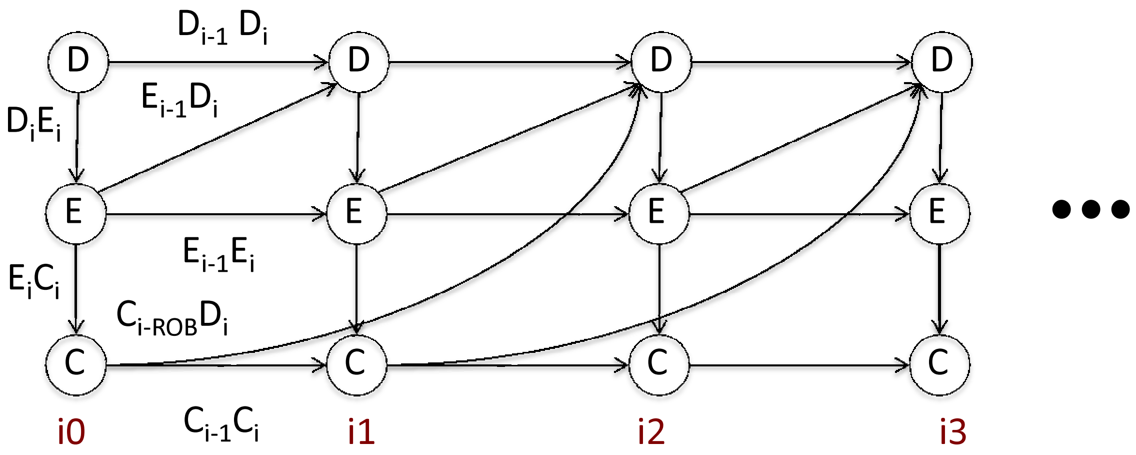


Figure 27: The dependence-graph model with four instructions.

Dependency	Source	Weight
$D_{i-1}D_i$	Limited dispatch-width	0/1/N (N: Instruction cache miss latency)
$C_{i-1}C_i$	Limited commit-width	0/1
D_iE_i	Dispatch-stage → execute-stage	1
E_iC_i	Execute-stage → commit-stage	$Lat_i + 1$ (Lat_{EC}: execution latency of instruction i)
E_xE_y	Data/memory dependency	Lat_x (Lat_{EE}: execution latency of instruction x)
$E_{i-1}D_i$	Branch misprediction	Lat (Lat_{ED}: latency required to resolve a branch misprediction from instruction $i-1$)
$C_{i-ROBsize}D_i$	Limited ROB size	0 ($ROBsize$: the ROB size)

Table 12: The dependencies (edges) depicted in the dependence-graph model.

one clock cycle plus instruction i's execution latency. The edge $\mathsf{E}_x\mathsf{E}_y$ represents the data dependency or memory dependency between two different instructions x and y. The weight of the $\mathsf{E}_x\mathsf{E}_y$ edge is the execution latency of instruction x. The edge $\mathsf{E}_{i-1}\mathsf{D}_i$ represents the dependency between two consecutive instructions $i-1$ and i, where instruction $i-1$ is a mispredicted branch instruction. There exists a dependency because when a branch misprediction occurs the new instructions are fetched from the correct execution path after the branch misprediction is resolved. The weight of the $\mathsf{E}_{i-1}\mathsf{D}_i$ edge is the time to resolve

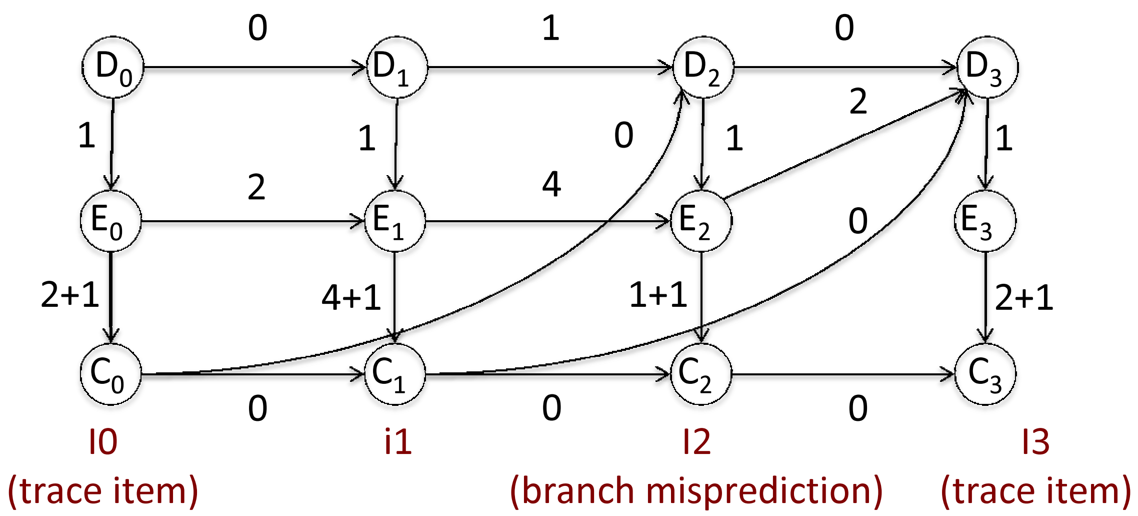


Figure 28: An example of collecting the abstract timing information using the dependence-graph model in trace generation. During trace generation, instruction 0 (i0) and 3 (i3) generate trace items and a branch misprediction occurs by instruction 2 (i2). Assume the ROB size is two and the L1 data cache hit latency is two cycles.

	instruction 0	instruction 1	instruction 2	instruction 3
dispatch time	1	1 (1+0)	5 (MAX(5+0, 1+1))	10 (MAX(5+0, 8+2, 9+0))
execute time	2 (1+1)	4 (MAX(2+2, 1+1))	8 (MAX(4+4, 5+1))	11 (10+1)
commit time	5 (2+2+1)	9 (MAX(5+0, 4+4+1))	10 (MAX(9+0, 8+1+1))	14 (MAX(10+0, 11+2+1))

Table 13: The weight on edges between the nodes in the dependence-graph model. Assume instruction 0 was dispatched at clock cycle 1, and the instruction execution latency of instructions 0, 1, 2, and 3 are 1, 4, 1, and 2 cycles, respectively. The branch misprediction penalty is set to 2 cycles.

a branch misprediction. Finally, the weight of $\mathsf{C}_{i-ROBsize}\mathsf{D}_i$ edge is always 0. The limited ROB size creates a dependency between instruction $i - ROBsize$ and i when ROB becomes full. In such case, instruction i will enter the ROB in the same clock cycle when instruction $i - ROBsize$ is committed.

Using Figure 28 and Table 13, an example is provided to demonstrate how the dependence-graph model is used to collect the abstract timing information. In Figure 28, suppose instruction 0 (i0) and instruction 3 (i3) generate trace items and a branch misprediction occurs by instruction 2 (i2) during trace generation. The dependence-graph model shows that the two instructions (i0 and i3) have the following four microarchitectural dependencies.

- Microarchitectural dependency caused by the limited dispatch-width.
- Microarchitectural dependency caused by the limited commit-width.
- Microarchitectural dependency caused by the limited ROB size.
- Microarchitectural dependency caused by a branch misprediction.

During trace generation, the abstract timing model estimates the time when an instruction is dispatched in the ROB (*tg_dispatch_time*), the time when an instruction is executed (*tg_execute_time*), and the time when an instruction is committed and removed from the ROB (*tg_commit_time*)[2], as shown in Table 13. The estimated times are recorded in trace items and are used during trace simulation to indicate the distance (in cycles) between two trace items.

4.2.5 Trace generation algorithm in In-N-Out

The trace generation process is described below using instruction N ($\mathsf{instruction}_N$) as an example, and the trace generation algorithm is presented in Figure 29. During trace generation, it is assumed that the target processor has infinite number of FUs.

1. Access the instruction cache using the PC (program counter) address of $\mathsf{instruction}_N$. If a miss occurs, an instruction cache miss trace item is generated.
2. Collect the time when $\mathsf{instruction}_N$ is dispatched (*tg_dispatch_time*). The dispatch time is the larger of the current dispatch time and the time when $\mathsf{instruction}_{N-ROBsize}$ was committed, where $robSize$ is the size of the ROB. *tg_dispatch_time* is computed as,

$$tg\_dispatch\_time = MAX(tg\_dispatch\_time, (tg\_commit\_time \text{ of } \mathsf{instruction}_{N-ROBsize}))$$

[2] The three notations used to describe the dispatch time (*tg_dispatch_time*), execute time (*tg_execute_time*), and commit time (*tg_commit_time*) of an instruction during trace generation are used throughout the rest of Section 4.

3. Decode $\textsf{instruction}_N$ and collect its dependency information. If $\textsf{instruction}_N$ depends on a preceding trace item, the ISN of the preceding trace item is recorded in the output register of $\textsf{instruction}_N$. The recorded ISN of the preceding trace item is propagated to the instructions depending on $\textsf{instruction}_N$ as shown in Figure 25.
4. Collect the time when $\textsf{instruction}_N$ is ready to execute (*tg_execute_time*). The time when $\textsf{instruction}_N$ is ready to execute is the larger of *tg_dispatch_time* +1 and the time when $\textsf{instruction}_N$'s dependency is resolved. *tg_execute_time* is computed as,

$$\textit{tg\_execute\_time} = MAX((\textit{tg\_dispatch\_time} + 1), \text{dependency resolve time})$$

5. If $\textsf{instruction}_N$ is a memory instruction (load or store instruction), the data cache is accessed. If a data cache miss occurs, a trace item is generated.
6. Collect the time when $\textsf{instruction}_N$ commits (*tg_commit_time*). The time when $\textsf{instruction}_N$ is committed is the larger of *tg_commit_time* and the time when $\textsf{instruction}_N$'s execution completes. *tg_commit_time* is computed as,

$$\begin{aligned} \textit{tg\_commit\_time} = MAX(\textit{tg\_commit\_time}, \\ (\textit{tg\_execute\_time} + \textsf{instruction}_N\text{'s execution latency} + 1)). \end{aligned}$$

A reduced trace item, i.e., L1 filtered trace item, captures the following information: (1) the ISN of the corresponding instruction, (2) the dependency information, (3) the abstract timing information (*tg_dispatch_time*, *tg_execute_time*, and *tg_commit_time*), (4) the cache access information, such as cache access type (data read, data write, instruction fetch), cache address, and writeback address (if a writeback occurs on a cache miss).

The details of the trace simulation are given in the following section.

```
/* n_dispatch (n_commit): the number of dispatched (committed) instructions in trace generation */
/* ROB_head_commit_time: tg_commit_time of $\mathsf{instruction}_{N-ROBsize}$ when $\mathsf{instruction}_N$ is currently simulated. $\mathsf{instruction}_N$ represents the $N_{th}$ instruction and $robSize$ represents the size of the ROB */

tg_dispatch_time = 0; tg_commit_time = 0;
while (there are instructions left to simulate) do
  Access instruction cache. Generate trace on an instruction cache miss.;
  n_dispatch++; n_commit++;
  if (ROB_head_commit_time > tg_dispatch_time) then
    tg_dispatch_time = ROB_head_commit_time; n_dispatch = 1;
  end if
  Identify all existing dependencies of the simulated instruction.;
  tg_ready_time = MAX(tg_dispatch_time +1, dependency resolve time);
  if instructions is a load/store instruction then
    Access data cache. Generate trace on a data cache miss.;
  end if
  Record the dependency information in the output register.;
  if ((n_dispatch % DISPATCH-WIDTH) == 0) then
    tg_dispatch_time++; n_dispatch = 0;
  end if
  inst_commit_time = tg_ready_time + instruction execution latency +1;
  if (inst_commit_time > tg_commit_time) then
    tg_commit_time = inst_commit_time; n_commit = 1;
  else if ((n_commit % COMMIT-WIDTH) == 0) then
    tg_commit_time++; n_commit = 0;
  end if
end while
```

Figure 29: The high-level pseudo-code of the trace generation algorithm.

4.3 TRACE SIMULATION IN IN-N-OUT

At the heart of out-of-order trace simulation is the ROB occupancy analysis introduced in Section 3.4.3. This section first revisits the ROB occupancy analysis, and then describes the trace simulation algorithm.

4.3.1 ROB occupancy analysis in In-N-Out

The ROB occupancy analysis described in Section 3.4.3 is also used in In-N-Out. The ROB occupancy analysis is simpler than PDCM, because trace items are always generated in program order and there are less trace items to analyze. Unlike PDCM, in In-N-Out, a delayed hit trace item is generated only if it has an effect on the ROB occupancy analysis. Figure 30 shows an example with five trace items and a 96-entry ROB.

In Figure 30(a), suppose all five trace items (L1 cache misses) A, B, C, D, and E miss in the L2 cache, and A is the head of the ROB. Given their ISN and the dependency information, only C can access the main memory in parallel with A. B can issue a cache access after A returns from the L2 cache. After A commits, the instruction in D can advance to the ROB as instructions between A and B commit. Instruction in E can move into the ROB after B commits. During trace generation in PDCM, a cache miss and the subsequent accesses to the same cache block (delayed hits) may not occur in program order. For instance, in PDCM, assume B-dh (instruction 120) and B (instruction 130) access the same L1 cache block, but B accessed the L1 cache before B-dh, and B-dh became a delayed hit because of B. If a trace item is not generated on a delayed hit, trace item D will move into the ROB after A commits. However, this is not correct because B-dh (instruction 120) will block the instructions committing from the ROB and ROB will be filled up without D (instruction 220). In In-N-Out, since In-N-Out uses an abstract timing model based on a functional simulator to generate traces, such problem does not occur because L1 cache misses happen in program order during trace generation.

Nevertheless, there is a case when a delayed hit trace item is needed in In-N-Out. The case is illustrated in Figure 30(b). Since store instructions do not wait for data from the

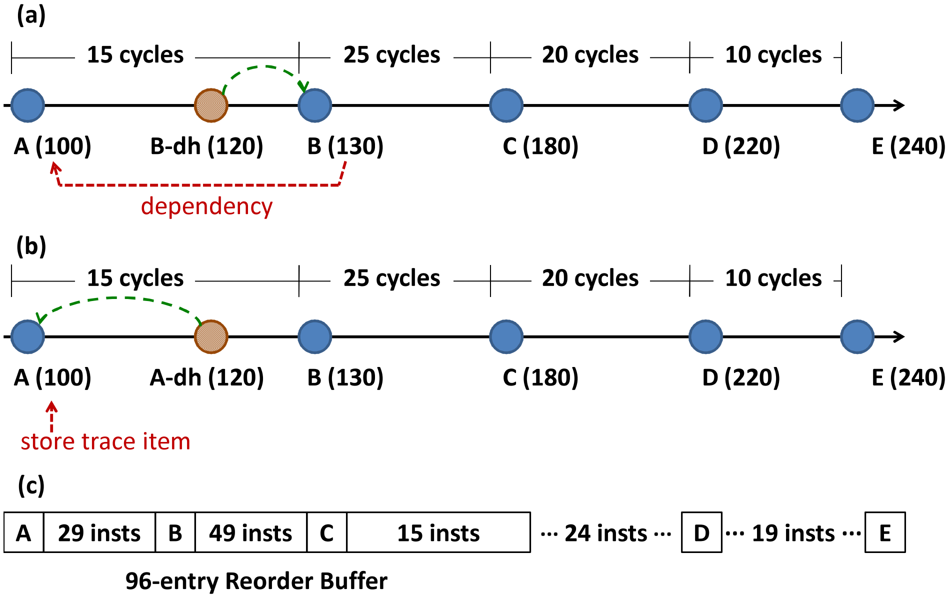


Figure 30: (a) Five L1 cache misses (A, B, C, D, and E) and one L1 delayed hit (B-dh) created by L1 cache miss B. Inside parentheses is the ISN of a memory instruction. (b) Assume L1 delayed hit (A-dh) is created by L1 cache miss A, and A is a write L1 cache miss from instruction 100. (c) The ROB occupancy status.

lower-level cache or the main memory, store instructions can be removed from the ROB right after accessing the data cache in the commit pipeline stage. Assuming a write-back and a write-allocate cache, if a store instruction misses in L1 and L2 caches, the subsequent load instructions that access the same cache block as the store instruction has to wait until the missed cache block is allocated. Assume memory instruction 120 is an L1 delayed hit created by the write L1 cache miss trace item A. Since instruction 120 has to wait until the missed cache block gets allocated, it will eventually become the head of the ROB and the ROB will be filled up. Then trace item D will be blocked from entering the ROB while the instruction 120 is the head of the ROB. To model such case in In-N-Out, a trace item is generated for the first read delayed hit following a write miss in the data cache. The trace items for the successive delayed hits after the first read delayed hit are not needed, since all delayed hits will be resolved at the same time, and only the earliest delayed hit will have an impact on the ROB occupancy status.

ROBsize	The size of ROB
rob-list	The list of trace items sorted in terms of the trace item's ISN
robHead	The trace item in the head of *rob-list*
sim_time	The current clock cycle time in trace simulation
dispatch_time	The estimated dispatch time of a trace item in trace simulation
ready_time	The time when a trace item is ready to be processed in trace simulation
return_time	The time when the trace item's cache access returns in trace simulation
issue-list	The list of trace items sorted in terms of the trace item's *ready_time*
issueHead	The trace item in the head of *issue-list*
dep_resolve_time	The largest parent trace item's *return_time* plus the recorded distance to the parent trace item
trace_process_time	The time to process *issue_head* in trace simulation
rob_head_commit_time	The time to remove *robHead* from *rob-list*

Table 14: Notations used for the In-N-Out algorithm description.

4.3.2 Simulation algorithm of In-N-Out

Table 14 list the notations used throughout the rest of this section. In trace simulation, two important lists—*rob-list* and *issue-list*—are employed to implement the simulation algorithm similar to PDCM. *rob-list* links the trace items in program order to reconstruct the ROB state during trace simulation. Trace items are inserted in *rob-list* if the difference between the trace item's *ISN* and *robHead*'s *ISN* is smaller than the ROB size. *issue-list* is used to process trace items out of order. Modern superscalar processors can issue instructions while long latency operations are still pending, if they are in the ROB and have no unresolved dependency. Similarly, the model determines that a trace item is ready to be processed, if it is in *rob-list* and has no unresolved dependency with other preceding trace items in *rob-list*. Ready trace items are inserted in *issue-list* and lined up with respect to their *ready_time*. The head of *issue-list* is always the one that gets processed. *issue-list* and *rob-list* are used to mimic the superscalar processor's ability to issue instructions out of order and commit completed instructions in program order. *rob-list* stalls the trace simulation when there are no trace items to process and new trace items are not inserted. The trace simulation resumes when new trace items are inserted after *robHead* is removed. This reflects

```
while (1) do
  sim_time++;
  if (sim_time == rob_head_commit_time) then
    Commit_Trace_Items();
    Update_ROB();
    update rob_head_commit_time for the new robHead
  end if
  if (sim_time == trace_process_time) then
    Process_Trace_Items();
  end if
  if (no more trace items left in the trace file) then
    break; /* END OF TRACE SIMULATION */
  end if
end while
```

Figure 31: The high-level pseudo-code of the trace simulation algorithm.

how a superscalar processor stalls the program execution when the head of the ROB is a pending memory instruction and there are no instructions to issue in the ROB. The processor resumes executing the program after the memory instruction commits and new instructions are dispatched into the ROB. In In-N-Out, since trace items (L1 data cache misses) are always generated in program order, *rob-list* does not need to keep the pending trace items as it did in PDCM (see Section 3.4.4.1).

Figure 31 presents the high-level pseudo-code of the trace simulation algorithm to model the superscalar processor with the baseline configuration described in Table 6. The key steps in the algorithm are described below.

Commit_Trace_Items. The instruction commit process employed in In-N-Out (shown in Figure 32) is similar to the process used in PDCM. *robHead* is removed from *rob-list*, if *sim_time* is larger than *robHead.return_time* (line 2). If *robHead* is generated by a store instruction, a write access to the L2 cache is issued before *robHead* is removed (lines 3 to 9). Since the trace items in *rob-list* may depend on *robHead* (the store instruction) via a delayed hit, dependent trace items in *rob-list* are searched after write access occurs (lines 5 to 8). If a dependent trace item is identified and all the dependencies of the dependent trace item are resolved, the dependent trace item's *ready_time* is set and it is inserted in *issue-list*.

```
1:  robNode = NULL;
2:  while (sim_time > (robHead.return_time +1)) do
3:    if (robHead is a write trace item) then
4:      Issue a write access to L2 cache;
5:      Resolve dependency for the trace items (delayed hits) that depend on robHead;
6:      if (A dependent trace item is ready to issue) then
7:        Set the dependent trace item's ready_time;
8:        Insert the dependent trace item in issue-list;
9:      end if
10:   end if
11:   robNode = robHead.next; /*next trace item in rob-list*/
12:   robHead = robNode;
13: end while
```

Figure 32: High-level pseudo-code for committing trace items.

After *robHead* is removed, the next trace item in *rob-list* becomes the new *robHead* (lines 11 and 12). Note that depending on the specified commit-width and the number of memory ports in the processor system, more than one trace item behind *robHead* can be removed from *rob-list* in the same cycle. The time to remove the next *robHead* from *rob-list* is indicated by *rob_head_commit_time*, which is computed after updating the ROB occupancy status.

Update_ROB. After committing the old trace items in *rob-list*, the algorithm attempts to insert new trace items from a trace file to *rob-list*. During trace simulation, the ROB is reconstructed as shown in Figure 33. First, the algorithm checks whether the new trace item can enter the ROB by comparing the *ISN* of *robHead* and the new trace item (line 1). If the difference is smaller than the ROB size, the new trace item is inserted in *rob-list*, otherwise, the algorithm stops fetching trace items from the trace file. Since multiple trace items are inserted in *rob-list* simultaneously, the algorithm estimates when the new trace items are actually dispatched in the ROB (line 2).

To estimate the dispatch time (*dispatch_time*) of the new trace items during trace simulation, the algorithm uses *tg_dispatch_time* collected during trace generation. An example is shown in Figure 34 assuming a 96-entry ROB. The example has four trace items from instructions 10 (A), 60 (B), 120 (D), and 140 (E). Assume L2 cache misses occur from all trace

```
while (ROB is not full fetch a new trace item from a trace file) do
  Estimate when the new trace item is dispatched in the ROB.
  if (The new trace item has an unresolved dependency on a preceding trace item in rob-list) then
    Mark the dependency information between the trace items.
  else
    Set the new trace item's ready_time;
    Insert the new trace item in issue-list.
  end if
end while
```

Figure 33: High-level pseudo-code for updating the ROB.

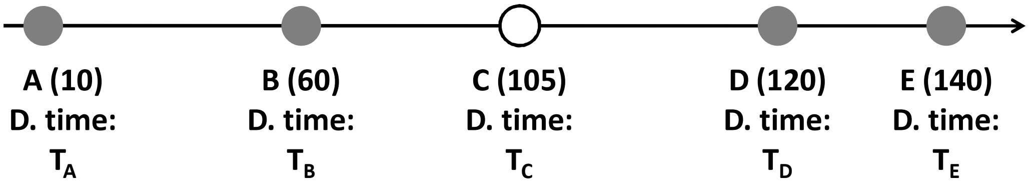


Figure 34: An example of estimating the dispatch time of trace items in trace simulation. Each circle represents an instruction. Circle filled in darkish color represents an instruction that generated a trace item. Inside parentheses is the ISN of an instruction. Instruction 105 (C) is the last (youngest) instruction in the ROB when instruction 10 (A) is the head of the ROB. "D. time" is the estimated dispatch time (*tg_dispatch_time*) of an instruction during trace generation.

items, and A was fetched at cycle N. The ROB will be filled while A waits for its requested data from the main memory with instruction 105 (C) being the last (youngest) instruction in the ROB. During trace simulation, *dispatch_time* of A and B is estimated as $N + T_A$ and $N + T_B$, respectively. After A commits, D advances in the ROB as instructions between A and B commit. *dispatch_time* of D is computed as $T_{commit-A} + (T_D - T_C)$, where $T_{commit-A}$ is the time when A is committed during trace simulation. Because the time spent dispatching instructions between B and C in the ROB is overlapped with the main memory access from A, using the time difference between D and B ($T_D - T_B$) to estimate *dispatch-time* of D would incur an incorrect delay. During trace generation, when instruction i creates an L1 cache miss (*trace_i*), *tg_dispatch_time* of instructions i and $i + ROBsize$ are recorded in the trace item generated from instruction i. Lastly, *dispatch_time* of E is computed as *dispatch_time* of D plus $T_E - T_D$.

Let us return back to the pseudo-code presented in Figure 33. If the new trace item depends on a preceding trace item in *rob-list*, the dependency information is marked in the two trace items (lines 3 and 4). If the new trace item has no dependencies to address, the trace item's *ready_time* is computed and inserted in *issue-list* (lines 5 to 6). The new trace item's *ready_time* is computed as

$$\begin{aligned} ready\_time = MAX(&\textit{dispatch\_time} + (\textit{tg\_execute\_time} - \textit{tg\_dispatch\_time}),\\ &\textit{dep\_resolve\_time}), (\textit{tg\_execute\_time} - \textit{tg\_dispatch\_time} \geq 1) \end{aligned}$$

Since a trace item may depend on a long sequence of instructions that are not trace items, the difference between *tg_execute_time* and *tg_dispatch_time*, collected during trace generation, is used to capture the execution latencies of its parent instructions. The difference between *tg_execute_time* and *tg_dispatch_time* is at least 1, because the abstract timing model assumes a one cycle latency during trace generation when an instruction moves from the dispatch-stage to the execute-stage as described in Section 4.2.4.

Update rob_head_commit_time. After updating *rob-list*, *rob_head_commit_time* is computed to indicate the time to remove the new *robHead*. The new *rob_head_commit_time* is estimated as MAX(*robHead-commit-time*, (*robHead.return_time* +1)) for the new *robHead*, where *robHead-commit-time* is computed as below.

$$\begin{aligned} \textit{robHead-commit-time} = \textit{sim\_time} + \textit{robHead.tg\_commit\_time} -\\ \text{prev. } \textit{robHead.tg\_commit\_time} \end{aligned}$$

Process_Trace_Items. The time to process *issueHead* is indicated by *traceProcessTime*. If *issueHead* was generated by a load instruction, the algorithm makes a read access to the L2 cache and then searches for the dependent trace items in *rob-list*. If *issueHead* was generated by a store instruction, *issueHead*'s *return_time* is set to *sim_time* and a write access is issued when *issueHead* is removed from *rob-list*; i.e., when it commits.

```
 1: if (trace item is a delayed hit) then
 2:   if (the selected MSHR can hold the current trace item) then
 3:     process the trace item
 4:   else
 5:     insert the trace item back in issue-list with a new ready_time
 6:   end if
 7: else
 8:   if (if there is a free MSHR) then
 9:     mark the trace item as the owner of the free MSHR and process the trace item
10:   else
11:     insert the trace item back in issue-list with a new ready_time
12:   end if
13: end if
```

Figure 35: The high-level pseudo-code for MSHR allocation. A trace item is determined a delayed hit if there is an MSHR holding pending trace items with the same tag address (line 1).

4.3.3 Modeling various processor artifacts in In-N-Out

Modeling the L2 data prefetcher. Modeling the data prefetcher in L2 cache is straightforward. Since the trace items represent the L2 cache accesses, the prefetcher monitors the L2 cache accesses from the trace items and generates a prefetch request to the memory as necessary. Other than adding a model for the data prefetcher, no additional changes are needed in the trace simulation algorithm besides creating the interface between the simulator and the data prefetcher model.

Modeling L2 MSHRs. This dissertation assumes that an L2 MSHR can hold the L2 cache miss and the delayed hits to the same cache block. Since the number of outstanding L2 cache misses is now limited by the available MSHRs in the processor, MSHRs are examined before a cache access is issued, as described in Figure 35.

Modeling the instruction caching effect. To model the instruction caching effect, during trace generation, trace items are generated on L1 instruction cache misses. The penalties from instruction cache misses are accounted during trace simulation by stalling the simulation when an instruction cache miss trace item is encountered and if there are no trace items to process in the ROB, as shown in Figure 36. In "Update_ROB" function, the algorithm stops updating *rob-list* if it fetches a trace item generated from an instruction cache miss. The L2 cache is accessed with the instruction cache miss trace item, and the returned latency is accumulated (*icache_miss_delay*). New trace items are fetched from a trace file only when *icache_miss_delay* is 0. The experiments reveal that with this simple strategy In-N-Out can accurately predict the increased clock cycles due to instruction cache misses.

Modeling the branch prediction. To account for the effect of branch prediction, a branch predictor is used in the trace generator. The penalty caused by branch mispredictions are modeled during trace generation as described in Section 4.2.3. Hence, a trace item's *tg_dispatch_time* collected during trace generation would be larger with a branch predictor in the trace generator, compared to *tg_dispatch_time* collected without a branch predictor in the trace generator. During trace simulation, the algorithm exploits *tg_dispatch_time* of a trace item and the dependency between trace items created by branch mispredictions.

```
/* icache_miss_delay: the accumulated L2 cache access latencies from instruction cache miss
trace items. */
while (1) do
  sim_time++;
  if (icache_miss_delay) then
    icache_miss_delay--;
  end if
  if (sim_time >= rob_head_commit_time) then
    if (robHead == NULL) then
      if (icache_miss_delay == 0) then
        Update_ROB();
      end if
    else
      Commit_Trace_Items();
      if (icache_miss_delay == 0) then
        Update_ROB();
      end if
      update rob_head_commit_time for the new robHead
    end if
  end if
  if (sim_time == trace_process_time) then
    Process_Trace_Items();
  end if
  if (no more trace items left in the trace file) then
    break; /* END OF TRACE SIMULATION */
  end if
end while
```

Figure 36: The high-level pseudo-code of the trace simulation algorithm incorporating the instruction caching effect.

4.4 EXPERIMENTAL SETUP

Two different machine models are used in experiments, “baseline” and “realistic.” Table 6 lists the baseline and realistic configurations, which resembles the Intel Core 2 Duo processor [17]. The baseline model assumes perfect branch prediction and instruction caching, infinite MSHRs, and no data prefetching. The realistic model uses realistic branch prediction and instruction caching, and also incorporates L2 data prefetching and L2 MSHRs. For L2 data prefetching, a tagged prefetch [69], a sequential prefetching technique, and a stream-based prefetching [71] technique are implemented.

The baseline and realistic machine configurations are simulated with two simulators: `sim-outorder` [2] and the In-N-Out trace-driven simulator (“`In-N-Out`”). For comparison, `sim-outorder` is extended with L2 data prefetching and L2 MSHRs. `In-N-Out` implements the algorithm described in Section 4.3. To drive `In-N-Out` a trace generator is needed. This dissertation adapts `sim-cache`, a functional cache simulator [2], for trace generation. To simulate the impact of branch mispredictions on program execution time, a branch-predictor model is added in the trace generator.

All benchmarks from the SPEC2K suite are used in the following experiments. For each simulation, the initialization phase of the target program [68] is skipped, then caches are warmed up for 100M instructions. The next 1B instructions are simulated after warming up the caches.

To evaluate `In-N-Out`, *CPI error* and *relative CPI change* are used as the main metrics. The evaluation metrics and their definitions are described in Section 3.4.5.

4.5 EVALUATION RESULT

4.5.1 Accuracy of In-N-Out

This section comprehensively evaluates In-N-Out. First, the accuracy of `In-N-Out` using the baseline and realistic configurations is presented, followed by the evaluation of `In-N-Out` in

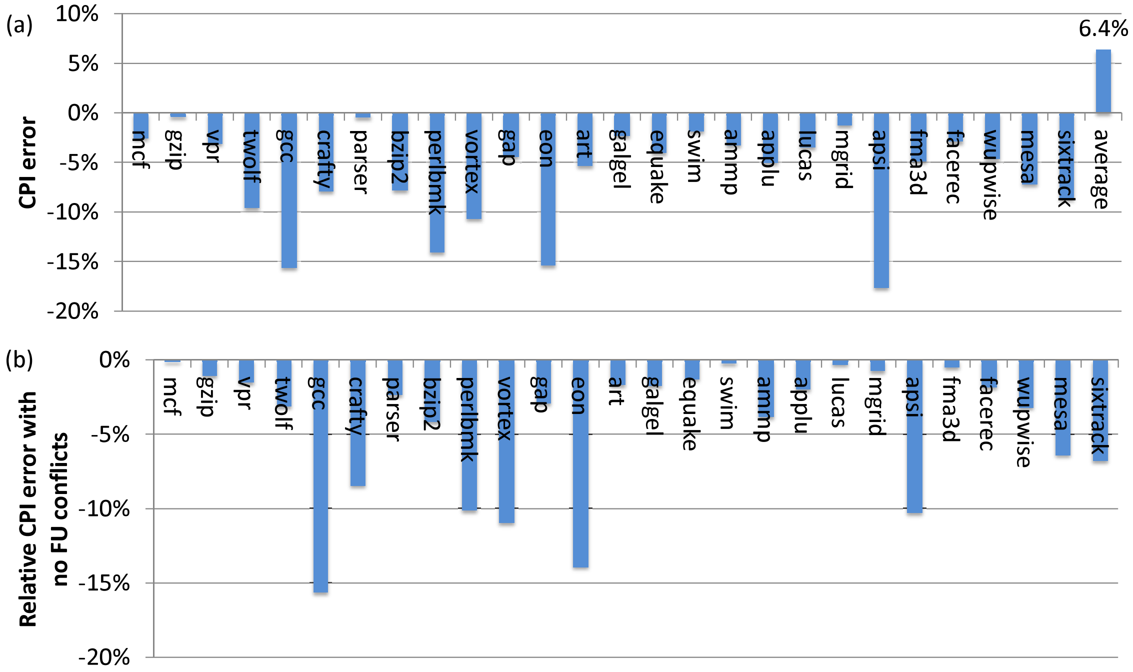


Figure 37: (a) The CPI errors of `In-N-Out` with the baseline configuration. (b) The relative CPI changes shown by `sim-outorder` when there are no FU conflicts.

terms of its uncore access behavior.

- **In-N-Out with the Baseline configuration.** `In-N-Out` is first evaluated with the baseline configuration. Figure 37(a) shows the CPI errors of the 26 SPEC2K benchmarks using the baseline configuration. The CPI errors range from −18% (*apsi*) to 0% (*gzip*) with an average of 6.4%. There are two main sources of error in `In-N-Out`. The first source of error is created by not modeling the penalties caused by the conflicts on functional units (FUs). FU conflict occurs if the number of operations accessing the same type of FU is larger than the number of available FUs in a given cycle. As a result, `In-N-Out` shows large CPI errors for the benchmarks that are sensitive to the number of FUs in the processor. Figure 37(b) shows the relative CPI change observed by `sim-outorder` when there are no FU conflicts. Benchmarks that show a relatively large CPI change, such as *gcc*, *perl*, *vortex*, *eon*, and *apsi*, also show a large CPI error. The second source of error is created by not modeling the effect of the instruction scheduling policy on program execution time. `In-N-Out` can be taken as a processor model that has no limit on the number of instructions it can can issue

Different ROB sizes				
size	32	64	128	256
Avg. CPI error	4.1%	5.8%	7.3%	8.6%

Different L1 data cache sizes			
size	8KB	16KB	64KB
Avg. CPI error	6.1%	6.4%	6.6%

Different dispatch-width (commit-width)		
width	2	8
Avg. CPI error	12.4%	8.5%

Table 15: The accuracy of `In-N-Out` with different processor core configurations.

per cycle, whereas `sim-outorder` schedules instructions according to a certain instruction scheduling policy due to the limited number of instructions it can issue per cycle. *twolf* is not sensitive to FU conflicts but it shows a relatively large CPI error (−10%), because in `sim-outorder`, the scheduling of memory instructions are frequently deferred by other instructions, but `In-N-Out` does not show such delays caused by the instruction scheduling policy.

Similar to `PDCM`, `In-N-Out` is robust to the variation in processor's inherent parameters. Different ROB size, L1 data cache size, and the processor's dispatch-width are used to study the sensitivity of `In-N-Out`. Table 15 summarizes the studied results. The results show that the accuracy of `In-N-Out` improves when smaller ROB is used. This is because smaller ROB reduces the amount of instruction-level parallelism (ILP), which reduces the number of FU conflicts. Accordingly, the accuracy of `In-N-Out` degrades when larger ROB is used, because the number of FU conflicts increases. The accuracy of `In-N-Out` slightly improves when a smaller data cache is used during trace generation. With smaller data cache, more trace items are generated during trace generation. In trace simulation, having more trace items help analyzing the ROB occupancy status, which helps improving the accuracy of `In-N-Out`. Finally, the processor's dispatch-width is used to examine the robustness of `In-N-Out`. In the experiments, the processor's commit-width was identical the processor's dispatch-width. In `sim-outorder`, the processor's issue-width was also same as the processor's dispatch-width.

As the processor's dispatch-width was changed from 4 to 2 and 8, more FU conflicts were observed from the benchmarks. Consequently, the accuracy of In-N-Out degraded as the processor's dispatch-width was changed from 4 to 2 and 8.

• **Effect of instruction caching in In-N-Out.** To model the effect of instruction caching in In-N-Out, trace items are generated from data and instruction cache misses during trace generation. For this experiment, a 32KB instruction cache, described in the realistic configuration 6, is employed in the baseline configuration.

The results show that incorporating the instruction caching artifact in In-N-Out does not affect the accuracy of In-N-Out. Similar to PDCM, only 7 benchmarks (out of 26), including *gcc*, *crafty*, *parser*, *perl*, *vortex*, *eon*, and *apsi*, showed a relative CPI change larger than 0%. The relative CPI difference of the 7 benchmarks was 1.1% on average and the largest relative CPI change was shown by *perl* from both sim-outorder (14%) and In-N-Out (12%). The CPI error using all 26 SPEC2K benchmarks was 6.5% on average. The results show that incorporating the instruction caching artifact in In-N-Out does not affect the accuracy of In-N-Out.

• **Effect of branch prediction in In-N-Out.** To examine how In-N-Out performs with a realistic branch predictor, a combined branch predictor (bimodal and gshare), described in Table 6, is incorporated in the baseline configuration. sim-outorder is configured with the identical branch predictor.

Figure 38 compares the CPI errors before and after incorporating a realistic branch predictor to the baseline configuration. The results show that incorporating the branch prediction artifact in In-N-Out does not largely affect the accuracy of In-N-Out, except for *gzip* and *eon*. *gzip* shows a CPI error close to 0% with the baseline configuration, however, when a branch predictor is added to the baseline configuration, *gzip* shows a large CPI error (-20%). The large CPI error of *gzip* was created by the large difference in branch misprediction counts between sim-outorder and the abstract timing model used to generate traces for In-N-Out. Since the abstract timing model is implemented on sim-cache, a functional cache simulator, the branch predictor is updated right after a branch prediction is simulated. However, in sim-outorder, the branch prediction occurs in the instruction fetch stage and then the branch predictor was updated in the instruction commit stage. The branch predic-

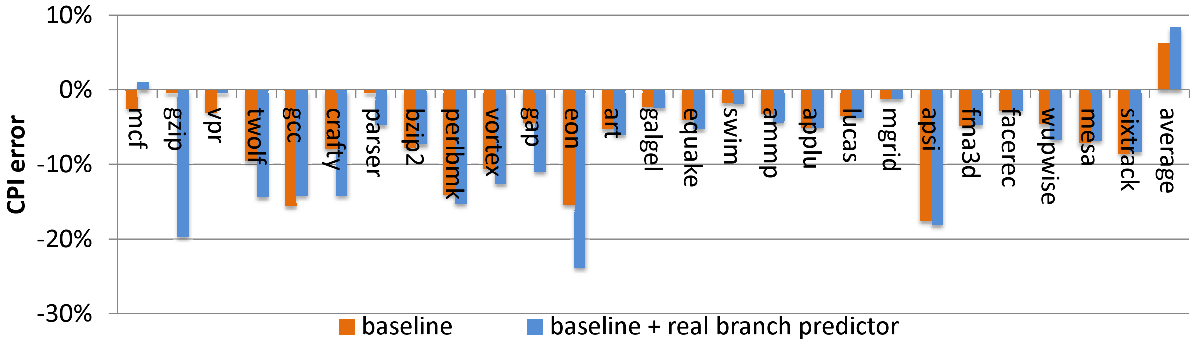


Figure 38: The CPI errors before (*base*) and after (*base* + *real branch predictor*) incorporating a realistic branch predictor in In-N-Out.

tion results of *gzip* and *eon* are affected by the difference in time when sim-outorder and sim-cache update the branch predictor. If the branch predictor is updated in the instruction dispatch stage in sim-outorder, the CPI error of *gzip* becomes −4%.

To evaluate how In-N-Out performs with regard to L2 data prefetching and L2 MSHRs, a relative metric is used to compare the CPI with and without these artifacts. To explore a large design space in early design stages, it is less critical to obtain very accurate (absolute) performance results of a target machine configuration. The performance model should rather quickly provide the performance change directions and amounts to correctly expose trade-offs among different configurations.

• **Effect of L2 data prefetching in In-N-Out.** Figure 39(a) and (b) compare the relative CPI change reported by sim-outorder and In-N-Out, when an L2 tagged prefetcher and an L2 stream prefetcher are added in the baseline configuration, respectively. The results show that In-N-Out can accurately model the effect of L2 data prefetching. When an L2 tagged prefetcher is employed, the two largest beneficiaries were *swim* and *mgrid* as shown by both sim-outorder and In-N-Out. When an L2 stream prefetcher is employed, the largest beneficiary was *fma3d* as shown by both sim-outorder and In-N-Out. Overall, In-N-Out closely follows the performance trend revealed by sim-outorder. The relative CPI differences of the relative CPI changes with the L2 tagged prefetcher and the L2 stream prefetcher were 1.2% and 1.8% on average, respectively. The CPI errors with the L2 tagged

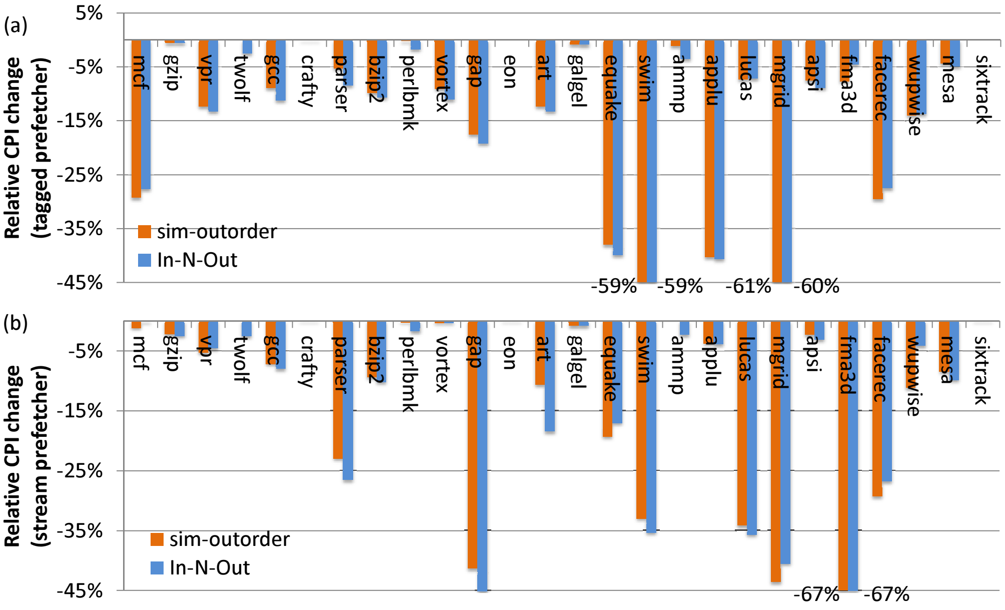


Figure 39: The relative CPI changes when different prefetching techniques are used, compared with no prefetching, in `In-N-Out`. For stream prefetching, the prefetcher's prefetch distance is 64 and the prefetch degree is 4, and 32 different streams are tracked in the prefetcher.

prefetcher and the L2 stream prefetcher in the baseline configuration were 6.9% and 7.5% on average, respectively.

• **Effect of limited L2 MSHRs in In-N-Out.** Figure 40 compares the relative CPI changes obtained with `In-N-Out` and `sim-outorder`, when limited number of L2 MSHRs is applied to the baseline configuration. Since the number of outstanding L2 cache misses is limited by the number of L2 MSHRs, the CPI increases with fewer MSHRs. The results show that `In-N-Out` can closely follow the relative CPI change of `sim-outorder`. As discussed in Section 3.4.6.1, *fma3d* is particularly sensitive to the number of MSHRs. `In-N-Out` was able to reproduce this unique behavior of *fma3d*. The largest relative CPI change was shown by *fma3d* with 4 MSHRs—316% and 333% with `sim-outorder` and `In-N-Out`, respectively. The average relative CPI difference of the relative CPI changes shown in Figure 40 was 1.7%. The average CPI errors after incorporating 4, 8, and 16 L2 MSHRs in the baseline configuration were 6.0%, 6.1%, and 6.2%. Finally, the accuracy of `In-N-Out` is reported below when the realistic instruction cache, realistic branch predictor, L2 MSHRs, and L2

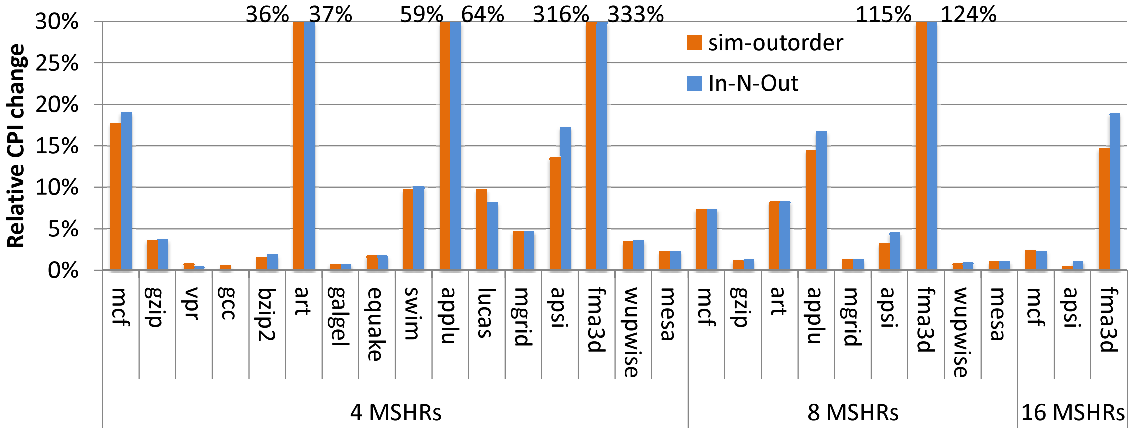


Figure 40: The relative CPI changes when 4, 8, and 16 MSHRs are used, compared with unlimited MSHRs, in `In-N-Out`. Among the entire 26 SPEC2K benchmarks, only the results of the benchmarks that showed at least 1% relative CPI change from either `sim-outorder` or `In-N-Out` are presented.

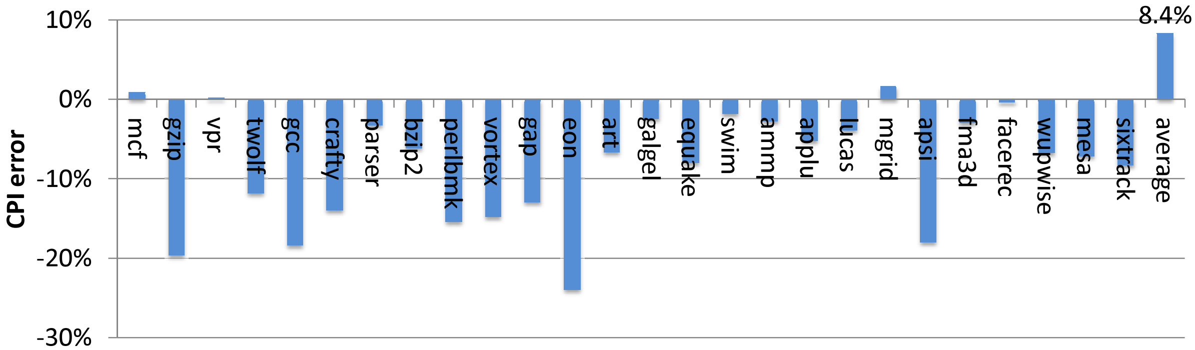


Figure 41: The CPI errors of the SPEC2K benchmarks using the realistic configuration in `In-N-Out`.

data prefetcher are added to the baseline configuration.

• **In-N-Out with the realistic configuration.**: The CPI error with the realistic configuration is presented in Figure 41. Comparing Figure 37 and Figure 41, the results show that `In-N-Out` maintains the average CPI error of the baseline configuration even when the realistic configuration is used. `In-N-Out` is also evaluated with a series of CPI errors measured over the program execution. The program execution is divided by an interval of 1M instructions. In each interval, CPIs are measured using `sim-outorder` and `In-N-Out` to

Benchmark	Avg.	Min.	Max.	Benchmark	Avg.	Min.	Max.
mcf	0.8%	0.0%	2.3%	*art*	8.0%	0.3%	17.9%
gzip	26.1%	15.2%	38.5%	*galgel*	4.2%	0.3%	30.1%
vpr	6.6%	0.0%	22.5%	*equake*	10.7%	0.3%	18.4%
twolf	13.5%	12.6%	14.4%	*swim*	2.7%	5.1%	0.8%
gcc	23.8%	0.1%	33.4%	*ammp*	6.7%	2.7%	11.1%
crafty	18.0%	13.5%	21.7%	*applu*	10.7%	1.2%	18.0%
parser	11.0%	0.1%	66.9%	*lucas*	5.6%	0.0%	14.5%
bzip2	8.2%	0.3%	22.9%	*mgrid*	1.8%	0.3%	18.5%
perl	21.7%	20.0%	23.5%	*apsi*	17.4%	1.6%	34.0%
vortex	17.8%	11.1%	20.0%	*fma3d*	10.7%	6.3.%	16.4%
gap	18.8%	0.8%	26.4%	*facerec*	1.6%	0.0%	3.7%
eon	28.3%	21.2%	32.8%	*wupwise*	8.6%	0.9%	14.3%
				mesa	10.6%	0.3%	12.7%
				sixtrack	15.5%	14.9%	15.2%

Table 16: The average, minimum, and maximum CPI errors of `In-N-Out` observed throughout a program execution using the realistic configuration.

compute the CPI error of `In-N-Out`. Table 16 shows the average, minimum, and maximum CPI errors of `In-N-Out` that were observed from 1,000 intervals using the entire SPEC2K benchmarks.

Similar to `PDCM`, the results show that there are benchmarks that show a large CPI error at some point during program execution. However, both `In-N-Out` and `sim-outorder` showed a very similar CPI trend while simulating 1B instructions. Figure 42 presents an example using *parser*, which showed the largest CPI error in an interval (66.9%) among all SPEC2K benchmarks. The 66.9% CPI error was observed in interval 535, where the CPI measured by `sim-outorder` was 0.75 and the CPI measured by `In-N-Out` was 1.25. Overall, the experiment results show that `In-N-Out` can closely follow changes in CPI over program execution shown by `sim-outorder`. Some benchmarks experience relatively large CPI errors from `In-N-Out` during trace simulation, however, it does not affect the overall trend.

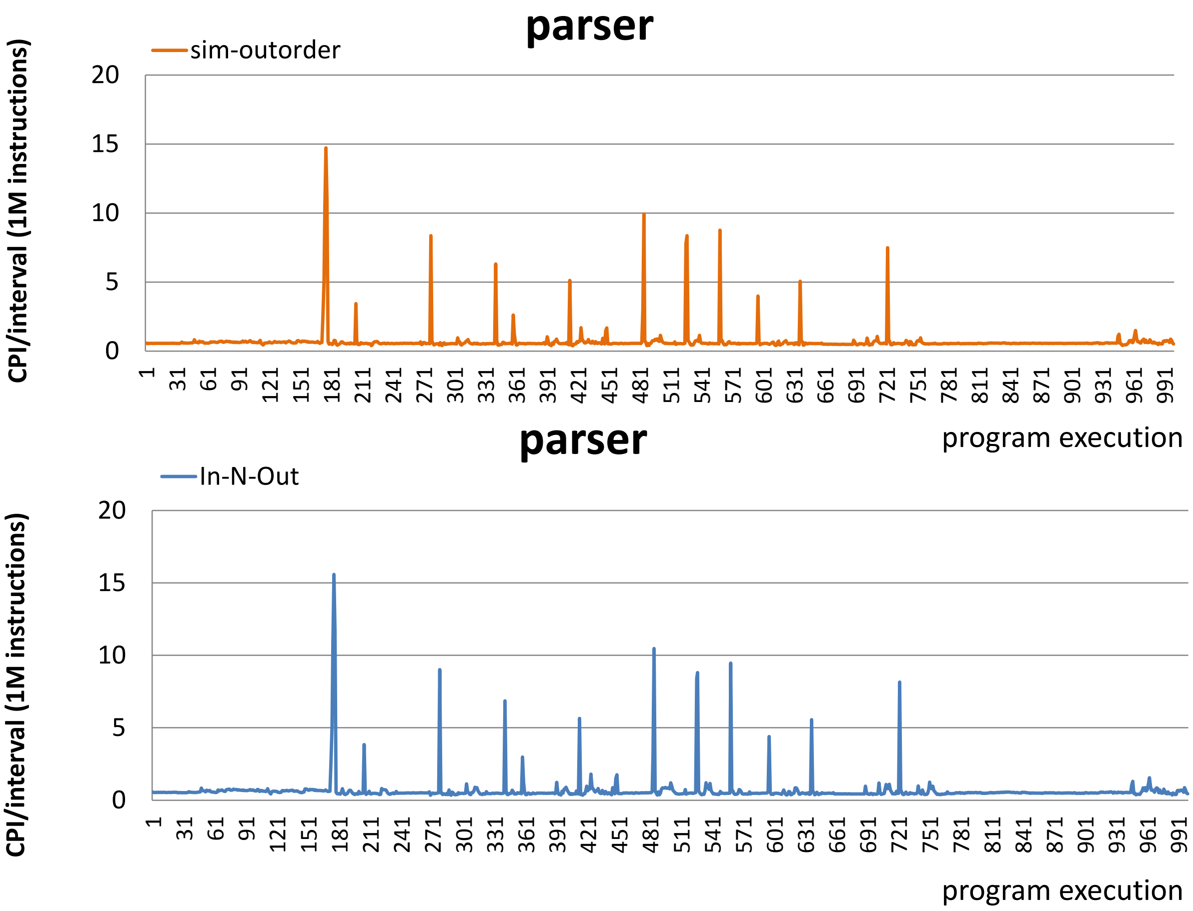


Figure 42: The CPI change of *parser* shown by `sim-outorder` and `In-N-Out` while simulating 1B instructions (1,000 intervals). Two separate figures are shown because it is difficult to observe the result due to the overlapped lines drawn from `sim-outorder` and `In-N-Out`.

4.5.2 Impact of uncore components

Until now, the evaluation of `In-N-Out` has focused on comparing the CPI of `sim-outorder` and `In-N-Out` by measuring the CPI error or the relative CPI change. In this subsection two important questions about how well `In-N-Out` captures the interaction of a processor core and its uncore components are addressed: (1) Does `In-N-Out` faithfully reproduce how a processor core exercises uncore resources? and (2) Can `In-N-Out` correctly reflect changes in the uncore resource parameters in the measured performance? These questions are especially relevant when validating the proposed In-N-Out approach in the context of multicore simulation; the shared uncore resources in a multicore processor are subject to contention as they are exercised and present variable latencies to the processor cores.

To explore the first question, for each benchmark, histograms of the distance (in cycles) between two consecutive off-chip accesses (from L2 cache misses, writebacks from L2 cache, or L2 data prefetching) are built over the program execution with `sim-outorder` and `In-N-Out`. The intuition is that if `In-N-Out` preserves the off-chip access patterns of `sim-outorder`, the two histograms should be similar. To track the temporal changes in a program, the program execution is first divided into intervals of 100M instructions and a histogram is generated for each interval. Each bin in a histogram represents a specific range of distances between two consecutive off-chip accesses. The value in a bin represents the frequency of distances that fall into the corresponding range. Since `In-N-Out` does not issue speculative off-chip accesses, in `sim-outorder`, only the distances between consecutive non-speculative off-chip accesses are collected.

To compare `sim-outorder` and `In-N-Out` with a single number, the *Similarity* metric introduced in Section 3.4.6.2 is used to evaluate `In-N-Out`.

$$Similarity = \frac{\sum_{i=0}^{n} \mathrm{MIN}(bin\_soo_i, bin\_io_i)}{\sum_{i=0}^{n} bin\_soo_i}$$

where i is the bin index and bin_soo_i and bin_io_i are the frequency value in i_{th} bin collected by `sim-outorder` and `In-N-Out`, respectively. The $MIN(bin\_soo_i, bin\_io_i)$ returns the common population between `sim-outorder` and `In-N-Out` in i_{th} bin. High similarity value implies `In-N-Out`'s ability to preserve the memory access pattern of `sim-outorder`. If the similarity is 1, it suggests that the frequency of the collected distances between off-chip accesses in the two simulators is identical. Table 17 presents the computed average *Similarity* over 10 intervals for all SPEC2K benchmarks, except *twolf*. All 25 examined benchmarks showed 90% or higher similarity. The *Similarity* metric was not applicable for *twolf*, because *twolf* did not have two consecutive off-chip accesses in an interval. `sim-outorder` showed only 2 off-chip accesses and `In-N-Out` showed 5 off-chip accesses while simulating *twolf*. `In-N-Out` issued all 5 off-chip accesses in the first interval ($0 - 100$M simulated instructions), however, `sim-outorder` showed one off-chip access in the first interval and the other off-chip access in the second interval ($100\text{M} - 200$M simulated instructions).

Figure 43 depicts the histograms of an interval of the benchmarks that show the lowest (*fma3d*) and highest (*mesa*) similarity for clear presentation. Only one interval of a benchmark is shown because most intervals of a benchmark show similar off-chip access patterns. *mesa* shows that `sim-outorder` and `In-N-Out` agree well on the off-chip access behavior, while *fma3d* shows that `sim-outorder` and `In-N-Out` disagree somewhat on the frequency of the distances between close off-chip accesses. In *fma3d*, some very short intervals (“0–12”) have shifted into the next, longer interval range (“13–24”). Overall, both plots show that `In-N-Out` preserves the temporal off-chip access patterns of the programs fairly well.

To address the second question, the relative CPI changes obtained with `sim-outorder` and `In-N-Out` are compared when five important uncore parameters are changed. Changing an uncore parameter makes the memory access latencies seen by the processor core different. The relative CPI differences are reported for all SPEC2K benchmarks across five new configurations in Table 18. An element in the table is the relative CPI difference between `sim-outorder` and `In-N-Out`. For example, when 2MB L2 cache is changed to 1MB L2 cache, *twolf* experiences a relative CPI change of 76% with `sim-outorder` and 85% with `In-N-Out`. The relative CPI difference of the two is 10% (rounded off), which is shown in the fifth row (*twolf*) and second column (Conf. 1) in Table 18. Note that the performance change directions predicted by `sim-outorder` and `In-N-Out` always agreed. The largest relative CPI difference was shown by *gcc* when the L2 cache size was reduced from 2MB to 1MB. The relative CPI change was 77% with `sim-outorder` and 91% with `In-N-Out`. Overall, the relative CPI differences were very small—the arithmetic mean of the relative CPI difference was under 2% for all five new configurations.

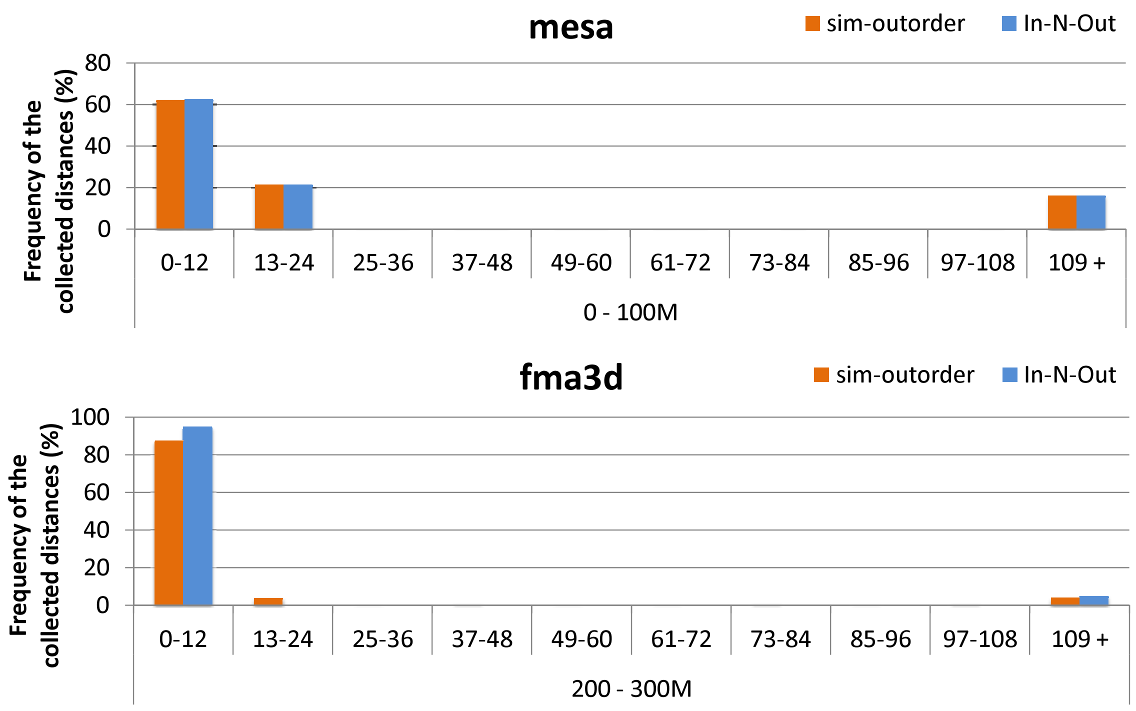


Figure 43: The histogram of collected distances between two consecutive memory accesses in `sim-outorder` and `In-N-Out` when executing *mesa* and *fma3d*. The x-axis represents the bins used to collect the distances and the y-axis represents the frequency of the collected distances in an interval of the program execution. The bin size is 12 cycles. Only one interval is shown as it is representative.

Similarity	Benchmark (similarity)
$< 95\%$	*fma3d*, *mgrid* (90%), *vortex* (92%), *equake* (93%)
	applu, *facerec*, *swim*, *wupwise* (94%)
$\geq 95\%$	*gcc* (95%), *art*, *gzip* (96%)
	ammp, *crafty*, *parser*, *vpr* (97%)
	apsi, *eon*, *galgel*, *lucas*, *perl* (98%)
	bzip2, *gap*, *mcf*, *sixtrack* (99%), *mesa* (100%)

Table 17: The similarity in memory access patterns between `sim-outorder` and `In-N-Out` (shown in percentage).

Benchmark	Conf. 1	Conf. 2	Conf. 3	Conf. 4	Conf. 5
mcf	1%	1%	0%	0%	0%
gzip	0%	0%	0%	0%	0%
vpr	0%	0%	0%	0%	1%
twolf	10%	0%	0%	0%	4%
gcc	14%	1%	3%	3%	1%
crafty	0%	0%	0%	0%	0%
parser	1%	1%	1%	1%	4%
bzip2	2%	1%	0%	0%	2%
perl	0%	0%	0%	0%	1%
vortex	0%	0%	1%	1%	0%
gap	0%	0%	2%	2%	1%
eon	0%	0%	0%	0%	0%
art	9%	6%	2%	2%	3%
galgel	0%	0%	0%	0%	1%
equake	0%	0%	2%	2%	1%
swim	0%	0%	1%	1%	0%
ammp	3%	0%	0%	0%	3%
applu	0%	0%	1%	2%	0%
lucas	0%	0%	1%	1%	0%
mgrid	0%	0%	5%	3%	0%
apsi	0%	0%	1%	1%	0%
fma3d	0%	0%	1%	2%	0%
facerec	3%	6%	0%	0%	1%
wupwise	0%	0%	1%	1%	0%
mesa	0%	0%	0%	0%	0%
sixtrack	0%	0%	0%	0%	0%
Avg. Error	1.7%	0.6%	0.9%	0.9%	1.0%

Table 18: The relative CPI differences between `sim-outorder` and `In-N-Out`. The five configurations are identical to the realistic configuration (Table 6) except a single parameter. In Configuration 1 (Conf. 1) and 2 (Conf. 2), the L2 cache is 1MB and 4MB instead of 2MB ("smaller L2 cache" and "larger L2 cache"). In Configuration 3 (Conf. 3) and 4 (Conf. 4), the memory latency is 100 cycles and 300 cycles instead of 200 cycles ("faster memory" and "slower memory"). In Configuration 5 (Conf. 5), the L2 hit latency is 20 cycles instead of 12 cycles ("slower L2 cache"). The performance change directions observed from the two simulators were identical.

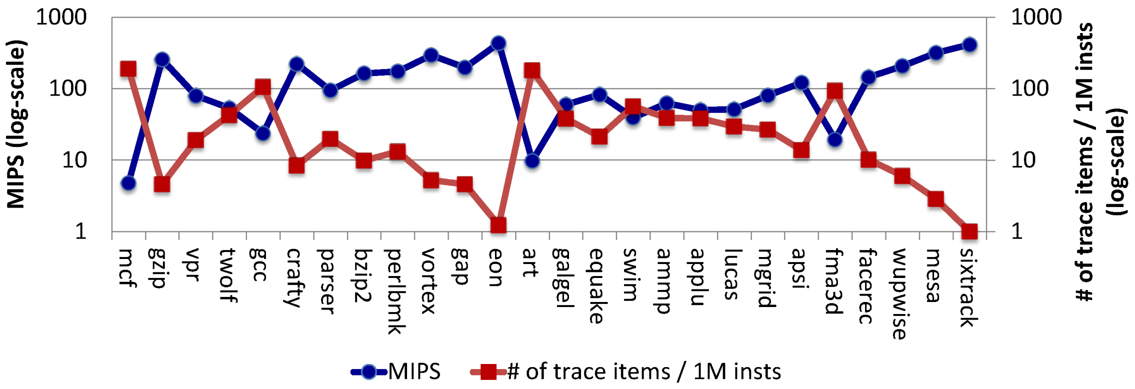


Figure 44: The relationship between the simulation speed and trace file size in In-N-Out.

4.5.3 Simulation speed and storage requirement

The biggest advantage of using In-N-Out over sim-outorder is its very fast simulation speed. The absolute simulation speed of In-N-Out and speedups over sim-outorder are measured with the realistic configuration on a 2.26GHz Xeon-based Linux box with an 8GB main memory. Similar to PDCM, In-N-Out's absolute simulation speed depends on the number of trace items to process as shown in Figure 44. The observed absolute simulation speeds range from 5 MIPS (*mcf*) to 434 MIPS (*eon*) and their average is 89 MIPS (geometric mean). The observed simulation speedups range from 15× (*art*) to 494× (*eon*) and their average (geometric mean) is 102×. Note that this is the actual simulation speedup without including the time spent for fast-forwarding in sim-outorder.

In-N-Out's absolute trace generation speed, using the realistic configuration, ranges from 1371 KIPS (*mcf*) to 1881 KIPS (*sixtrack*) and their average was 1709 KIPS (geometric mean). The trace generation speedups achieved with In-N-Out over PDCM using the realistic configuration range from 1.56× (*lucas*) to 3.97× (*gap*). The average (geometric mean) trace generation speedup was 2.26× on average.

A single trace item with In-N-Out was 24B, and the actual trace file size of the SPEC2K benchmarks was 24 (*sixtrack*) to 4,845 (*mcf*) in bytes per 1,000 simulated instructions. The trace file size can be further reduced by compressing the trace file when it is not used.

Note that trace file size reductions of over 70% are not uncommon when using well known compression tools like gzip.

4.6 SUMMARY

This chapter introduced In-N-Out, a novel trace-driven simulation strategy to evaluate out-of-order superscalar processor performance with reduced in-order traces. This chapter demonstrated that In-N-Out achieves reasonable accuracy in terms of absolute performance estimation, and more importantly, it can accurately predict the relative performance change when the uncore parameters such as L2 cache configuration are changed. In-N-Out can easily incorporate important processor artifacts such as data prefetching and MSHRs, and track the relative performance change caused by those artifacts. Compared with a detailed execution-driven simulation, In-N-Out achieves an absolute simulation speed of 89 MIPS on average (geometric mean) when running the SPEC2K benchmarks. This chapter concludes that In-N-Out provides a very practical and versatile framework for superscalar processor performance evaluation.

5.0 COMPARING PDCM AND IN-N-OUT

In Section 3 and 4, trace-driven simulation methods were introduced to model superscalar processor performance using reduced traces. In this section, the two signature simulation methods proposed in this dissertation, `PDCM` and `In-N-Out`, are compared.

First of all, both `PDCM` and `In-N-Out` use reduced trace, which only captures the uncore accesses. In terms of simulation accuracy, `PDCM` achieves higher accuracy than `In-N-Out` by exploiting the timing information collected during trace generation. To collect the timing information, `PDCM` employs a cycle-accurate timing simulator to generate reduced traces. On the other hand, `In-N-Out` cannot capture the correct timing information because it generates reduced traces using an abstract timing model based on a functional simulator or a binary instrumentation tool. However, `In-N-Out` can still provide reasonably accurate simulation results based on the abstract timing information and the dependency information between trace items. The reduced trace in `PDCM` may have speculated trace items generated from the mispredicted execution path of a program. On the other hand, the reduced trace in `In-N-Out` does not have any speculated trace items.

In this section, the evaluation results of `PDCM` and `In-N-Out` are first compared. Then, the result of a case study is reported to show how well `PDCM` and `In-N-Out` respond to different uncore configurations. Lastly, the limitations of `PDCM` and `In-N-Out` are discussed.

5.1 COMPARING THE ACCURACY

The absolute CPI error shows the amount of difference between the CPIs measured from `sim-outorder` and `PDCM` and `In-N-Out`. The machine configurations and the observed ab-

	Metric	Configuration	PDCM	In-N-Out
S1	CPI err.	baseline configuration (base cfg.)	1.9% (−6% ∼ 5%)	6.4% (−18% ∼ 0%)
S2	CPI err.	base cfg. & 8KB d-cache	2.5% (−6% ∼ 6%)	6.1% (−17% ∼ 0%)
S3	CPI err.	base cfg. & 16KB d-cache	2.1% (−6% ∼ 5%)	6.4% (−19% ∼ 0%)
S4	CPI err.	base cfg. & 64KB d-cache	1.8% (−5% ∼ 5%)	6.6% (−17% ∼ 0%)
S5	CPI err.	base cfg. & 2 disp-width	2.2% (−14% ∼ 7%)	12.4% (−24% ∼ −2%)
S6	CPI err.	base cfg. & 8 disp-width	2.1% (−6% ∼ 7%)	8.5% (−29% ∼ 1%)
S7	CPI err.	base cfg. & 32-ROB	1.5% (−13% ∼ 4%)	4.1% (−11% ∼ 0%)
S8	CPI err.	base cfg. & 64-ROB	1.9% (−17% ∼ 3%)	5.8% (−17% ∼ 0%)
S9	CPI err.	base cfg. & 128-ROB	2.3% (−11% ∼ 3%)	7.3% (−21% ∼ 0%)
S10	CPI err.	base cfg. & 256-ROB	2.8% (−10% ∼ 9%)	8.6% (−24% ∼ 0%)
S11	CPI err.	base cfg. & 32KB i-cache	1.8% (−6% ∼ 5%)	6.5% (−17% ∼ 0%)
S12	CPI err.	base cfg. & branch pred.	1.8% (−5% ∼ 4%)	8.4% (−24% ∼ 1%)
S13	CPI err.	realistic configuration	1.6% (−7% ∼ 7%)	8.3% (−24% ∼ 1%)

Table 19: Absolute CPI errors of PDCM and In-N-Out using different machine configurations.

solute CPI errors are presented in Table 19. The absolute CPI errors with the baseline configuration using PDCM and In-N-Out are 1.9% and 6.4% on average, respectively (S1). PDCM achieves high simulation accuracy, regardless of the processor core configuration, by exploiting the timing information recorded in the trace items. On the other hand, In-N-Out relies on abstract timing information, which causes larger CPI error than PDCM. In-N-Out achieves smaller CPI error for the programs that show regular memory access patterns, such as the floating point benchmarks in the SPEC2K benchmark suite, and shows larger CPI error for the programs that produce irregular memory access patterns with frequent branch predictions, such as the integer benchmarks in the SPEC2K benchmark suite. In-N-Out also shows larger CPI error for the benchmarks that experience frequent FU conflicts or frequent delays on issuing certain instructions due to the instruction scheduling policy. To examine the robustness of PDCM and In-N-Out to the variation in processor's inherent parameters, different L1 data cache sizes, instruction dispatch-width, and ROB sizes were used in the experiments.

The number of generated trace items increases when smaller L1 data cache is assumed during trace generation (S2 ∼ S4). In PDCM, the absolute CPI error tends to increase if the number of trace items increases because a timing error may occur when trace items are inserted in the ROB during trace simulation. On the other hand, in In-N-Out, the absolute CPI error tends to decrease if the number of trace items increases because more trace items help analyzing the ROB occupancy status more accurately during trace simulation.

The correlation between the processor's dispatch-width and the accuracy of PDCM and In-N-Out was not clear (S5 and S6). PDCM maintained small CPI errors regardless of the processor's dispatch-width. However, In-N-Out showed larger CPI error (on average) when the processor's dispatch-width was set to 2 and 8 instead of 4, because the integer benchmarks experienced more FU conflicts when the dispatch-width was changed from 4 to 2 and 8 in the baseline configuration. In the experiments, the number of FUs in the baseline configuration was changed accordingly when the processor's dispatch-width was changed. The ROB size determines the amount of instruction-level parallelism (ILP) a program can extract from the processor (S7 $\sim$ S10). Higher ILP is typically achieved with larger ROB in the processor, and lower ILP is shown with smaller ROB. The amount of ILP has an effect on the number of FU conflicts a program can experience, which affects the absolute accuracy of In-N-Out. With smaller ROB size, the processor limits the instruction-level parallelism, which reduces the number of FU conflicts. Hence, In-N-Out achieves smaller CPI error with smaller ROB size, but shows larger CPI error with larger ROB size. In PDCM, a timing error may occur when the trace simulation algorithm determines the time to insert a pending trace item in the ROB, especially when the distance between the last trace item in the ROB and the last instruction in the ROB is large. With smaller ROB, we observed that the trace items tend to fill up the ROB more frequently, which reduces the chances of introducing errors during trace simulation.

The branch prediction and instruction caching did not have a large affect on the CPI errors for both PDCM and In-N-Out (S11 and S12). However, there were a few benchmarks (*gzip* and *eon*) that showed large CPI errors in In-N-Out when a realistic branch predictor was incorporated in the baseline configuration. The branch prediction results of *gzip* and *eon* were affected by the difference in time when the branch predictor was updated in the abstract timing model, the trace generator used in In-N-Out, and sim-outorder.

Finally, the absolute CPI errors of the SPEC2K benchmarks between PDCM and In-N-Out are compared using the realistic configuration (S13). The realistic configuration combines the baseline configuration and the important processor artifacts, such as branch prediction, instruction caching, L2 data prefetching, and L2 MSHRs. When the realistic configuration is used, the accuracy of PDCM and In-N-Out depend on how accurately the processor artifacts are modeled in PDCM and In-N-Out. For instance, in In-N-Out, the CPI error of *gzip* is 0% when the baseline configuration is employed. However, the CPI error of *gzip* is −20% when the realistic configuration is used, which is caused by the errors from the branch mispredictions during trace generation.

Both PDCM and In-N-Out can accurately predict the relative performance change when uncore configurations are changed. The experiment results show that PDCM and In-N-Out closely follow the relative CPI changes predicted by sim-outorder. We note that the most sensitive benchmarks observed from sim-outorder, PDCM, and In-N-Out to the changes on uncore configurations were identical.

To explore whether PDCM and In-N-Out change how a processor core exercises the uncore resources, for each SPEC2K benchmark, the distances between two consecutive off-chip accesses were measured using PDCM, In-N-Out, and sim-outorder. A metric called "Similarity" was introduced to show how well PDCM and In-N-Out can reproduce the off-chip access patterns shown by sim-outorder. Overall, the experiment results showed that PDCM and In-N-Out show over 90% similarity on off-chip access patterns for most of the SPEC2K benchmarks.

5.2 CASE STUDY

The evaluation results presented in Section 3.4.6 and 4.5 strongly suggest that PDCM and In-N-Out offer adequate performance prediction accuracy for studies comparing different machine configurations. Moreover, the two simulators (PDCM and In-N-Out) were shown to successfully reproduce a superscalar processor's dynamic uncore access behavior. To further show the effectiveness of the two simulators, a case study is designed and conducted which involves L2 MSHRs, an L2 stream prefetcher, L2 cache associativity, L2 cache sizes, and a simple DRAM model. For this study, five different sets of programs are selected for each experiment. Each set has the eight most sensitive programs to the studied parameter.

When the number of MSHRs increases, the CPI decreases because more memory accesses can be outstanding simultaneously. The two simulators and `sim-outorder` reported the largest decrease in CPI when the number of MSHRs increased from 4 to 8 as shown in Figure 45(a). The CPI becomes stable when more MSHRs are provided. The close CPI change shown by the two simulators is a result of a good reproduction of the temporal memory access behavior of `sim-outorder`.

Figure 45(b) shows that when the L2 cache associativity is increased, the two simulators and `sim-outorder` reported the largest decrease in CPI when the L2 cache associativity changed from 1-way to 2-way associativity.

In a stream prefetcher, the larger (smaller) prefetch distance and prefetch degree makes the prefetcher more aggressive (conservative) when making prefetching decisions [71]. In general, the CPI increases if the prefetcher becomes more conservative. Figure 45(c) shows that the performance change predictions from the two simulators were less accurate when the stream prefetcher's configuration is changed, compared with the performance change predictions made using other uncore parameters. However, the results show that the two simulators can still be used to decide the configuration of the stream prefetcher. All three simulators reported the largest CPI increase when the prefetch distance and degree were changed from (16, 2) to (8, 1). All three simulators showed that the lowest and highest CPIs were observed when the stream prefetcher's prefetch distance and degree were set to (4, 1) and (64, 4), respectively. Different stream prefetcher configuration or different set-associativities have an effect on CPI by changing the number of cache misses during simulation. The close CPI trend, observed using `sim-outorder` and the two simulators, shows that our simulation methods can correctly follow how the core responds to different uncore access latencies (e.g., cache misses).

Figure 45(d) shows that when the L2 cache size is increased, the two simulators and `sim-outorder` reported the largest decrease in CPI when the L2 cache size was increased from 4MB to 8MB.

Lastly, instead of using a constant main memory access latency, in Figure 45(e), the two simulators are evaluated with various main memory access latencies using a simple DRAM model. A DRAM model that has 16 banks (8 banks x 2 ranks) with 16KB row size and an open-page policy were assumed for `sim-outorder`, `PDCM`, and `In-N-Out`. In the results, the three simulators show a linear increase in CPI when the page hit and miss latencies are increased linearly.

The results shown in the case study suggest that both `PDCM` and `In-N-Out` can be effectively used in the place of `sim-outorder` to study the relatively fine-grain configuration changes. Note that the performance change trend shown by `PDCM` was extremely close to `sim-outorder`.

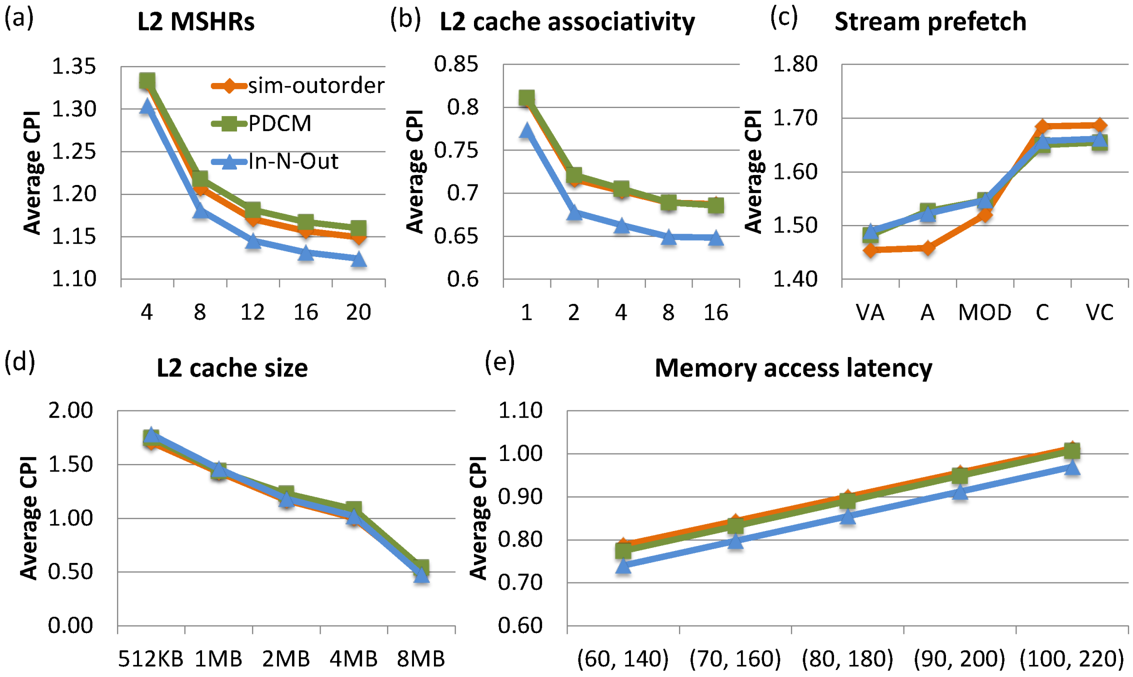


Figure 45: Comparing the trend in performance (average CPI) change of the superscalar processor between `sim-outorder` and `PDCM` and `In-N-Out`. The effects of L2 cache configuration and main memory access latency on performance are studied using the three simulators. The following changes have been made on the realistic configuration: (a) 5 different MSHRs: 4, 8, 12, 16, and 20 MSHRs. (b) 5 different L2 cache associativities: 1, 2, 4, 8, and 16. (c) 5 different stream prefetcher configurations (prefetch distance, prefetch degree). (d) 5 different L2 cache sizes: 512KB, 1MB, 2MB, 4MB, and 8MB. (e) 5 different DRAM model configurations (page hit latency, page miss latency). For each study, the top eight benchmarks that showed the largest performance change amount when observed with `sim-outorder` are used.

5.3 LIMITATIONS OF PDCM AND IN-N-OUT

Finally, the limitations of the PDCM and In-N-Out are discussed below.

- PDCM requires a detailed cycle-accurate simulator that models the target processor to generate timing-aware reduced traces. In-N-Out uses an abstract timing model implemented on top of a functional simulator or a binary instrumentation tool to make the trace generation process easier and faster than PDCM. However, In-N-Out does not provide as highly accurate simulation results as PDCM.
- PDCM and In-N-Out require an initial simulation to generate traces. PDCM is not practical when running a simulation one time only because it uses a detailed cycle-accurate simulator to generate traces. On the other hand, In-N-Out can still be practical to run a one-time simulation, because it is based on a functional simulator or a binary instrumentation tool to quickly generate traces. Note that the trace generation time is amortized as the generated traces are reused for different uncore configurations.
- The reduced traces have to be regenerated if a core parameter is changed. In this research, both PDCM and In-N-Out focus only on assessing the impact of uncore events on program execution time and assume that a superscalar processor core's parameters are fixed during a series of uncore experiments.

6.0 CONCLUSIONS

This dissertation presented practical trace-driven simulation methods that can quickly model the performance of superscalar processors using reduced traces. Conventional simulation methods for superscalar processors, such as execution-driven simulation and trace-driven simulation with full instruction traces, provide accurate simulation results. However, they are slow and storing a full instruction trace requires a large storage space. Compared with the conventional approaches, the presented methods achieve faster simulation speed while creating only a small error and use smaller disk space.

In this dissertation, two trace-driven superscalar processor simulation methods, pairwise dependent cache miss model (PDCM) and In-N-Out, are mainly discussed. The dissertation described how one can model the superscalar processor performance using reduced traces when the focus of study is on assessing the impact of uncore events, such as L1 cache misses, on program execution time. In PDCM and In-N-Out, important processor information is recorded in the reduced traces during trace generation, and then the recorded information is exploited during trace simulation to model superscalar processor performance.

The following contributions are made to the field of performance modeling in computer architecture.

- The dissertation proposed practical trace-driven simulation methods to quickly and accurately model a realistic superscalar processor performance. Unlike the conventional simulation methods for superscalar processors, the proposed methods use reduced traces and abstract a superscalar processor core's dynamic behavior to achieve fast simulation speed.
- The dissertation presented pairwise dependent cache miss model (PDCM), which enables

highly accurate trace simulation of superscalar processors using timing-aware reduced traces. The traces are generated from a cycle-accurate simulator. PDCM achieves an absolute simulation speed of 48 MIPS on average (geometric mean) while giving sufficiently small errors across benchmarks (less than 3% on average), when compared with a detailed execution-driven simulation method. The simulation speedup achieved by PDCM over a detailed execution-driven simulator was 55× on average.

- The dissertation presented In-N-Out, which achieves accurate trace simulation of superscalar processors using reduced inorder traces. The traces are generated from an abstract timing model implemented on top of a functional simulator or a binary instrumentation tool. Compared with PDCM, the trace generation and trace simulation is simpler and faster, however, the trace simulation results are less accurate. In-N-Out achieves an absolute simulation speed of 89 MIPS on average (geometric mean) while giving reasonably small errors across benchmarks (less than 7% on average), when compared with a detailed execution-driven simulation method. The simulation speedup achieved by In-N-Out over a detailed execution-driven simulator was 102× on average.
- Both PDCM and In-N-Out accurately predict the relative performance change using different uncore configurations. The performance change direction is always predicted correctly and the performance change amount is predicted with small errors. Moreover, both PDCM and In-N-Out are capable of faithfully replaying how a superscalar processor exercises and is affected by the uncore components.

Since this research uses a reduced trace-based simulation approach, there are a few limitations on the presented work. The limitations and the assumptions made in this dissertation are summarized below.

- The proposed trace-driven simulation methods focus only on assessing the impact of uncore events, such as L1 cache misses, on program execution time. Hence, they cannot be used for processor core simulation.
- Because the proposed methods require an initial simulation to generate traces, they are not practical when running a simulation one time only. However, the trace generation time will be amortized as the generated traces are reused.

- The reduced traces have to be regenerated if one or more processor core parameters are changed. However, the focus of this research is on assessing the impact of uncore events, such as misses on on-chip caches, on program execution time. Hence, it was assumed that a superscalar processor core's parameters are fixed during a series of uncore experiments.

Nevertheless, the presented simulation methods are attractive in the early processor design stages due to their fast simulation speed. Finally, I conclude that the two main simulation methods, PDCM and In-N-Out, presented in this dissertation provide a very practical and versatile framework for fast superscalar processor performance evaluation.

7.0 FUTURE RESEARCH DIRECTION

This thesis opens a new area on a simulation methodology research. The study demonstrates that trace-driven simulation with reduced trace is a promising approach. There are sill many interesting research topics that can be considered as the possible future work.

7.1 TRACE-DRIVEN SIMULATION FOR MULTI-CORE PROCESSORS

As current and future processor research is centered on multicore architectures, the importance of studying uncore components such as shared L2 cache, on-chip network, and memory controller, will continue to grow. However, modeling the performance of complex multicore systems with detailed cycle-accurate simulation is extremely time consuming. The situation is aggravated as large core counts are expected in future multicore systems. However, existing multicore simulators for superscalar processors [8, 51, 63] are not scalable to support large core counts. Moreover, they do not have sufficient simulation speed to conduct numerous studies in the early design stages. Given that the importance of simulation productivity will only grow with multicore scaling, multicore simulators must be faster and more scalable than today. I believe that PDCM and In-N-Out are the essential first step for developing a very fast and scalable multicore simulator that can model the performance of systems with many superscalar processor cores. In-N-Out uses a simpler and faster trace generation approach than PDCM, hence, I expect to have a novel and efficient multicore simulation environment for many superscalar processor cores by extending the In-N-Out framework. Tracing algorithm is required to collect traces from multithreaded programs, such as PARSEC [7] and SPLASH-2 [78].

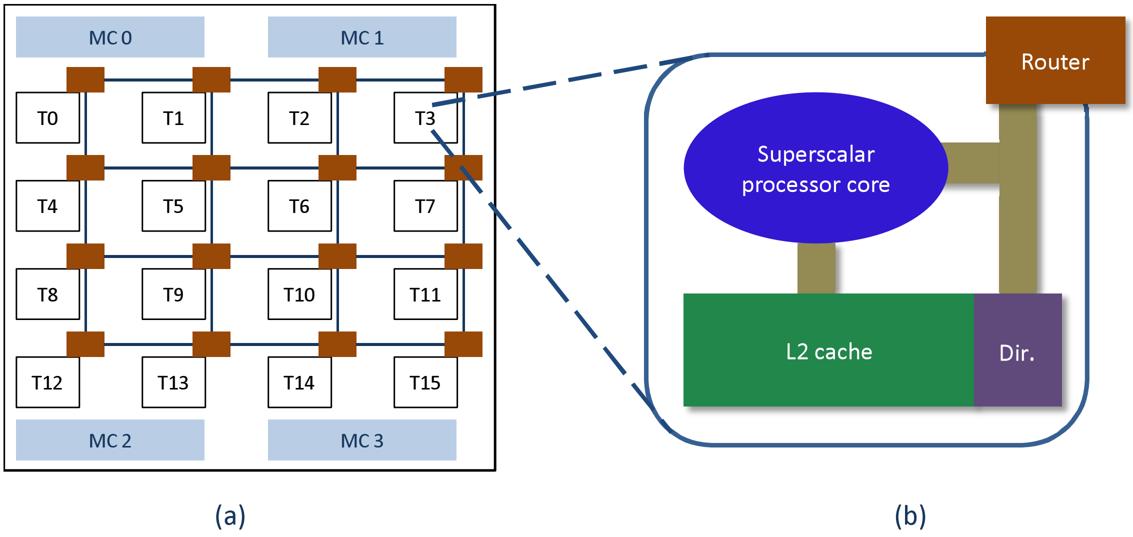


Figure 46: (a) A tile-based multicore system with 16 superscalar processor cores. (b) Each tile has a superscalar processor core (depicted in Figure 3), L2 cache slice, directory, and a router.

7.1.1 Multicore system model

The target multicore system for future research is a tile-based homogeneous chip multiprocessor (CMP) with a 2D mesh interconnection network. Figure 46(a) depicts an example of having 16 superscalar processors and Figure 46(b) shows the superscalar processor core and uncore components in each tile. L2 cache and directory are physically distributed in each tile. For future research, let us assume the multicore system model employs private L2 cache organization because private cache is expected to achieve better performance than shared cache in manycore processors [41]. Both L1 and L2 cache are write-back and write-allocate cache. Invalidation-based MESI protocol is assumed to maintain cache coherence.

7.1.2 Goals

There are two primary goals for the multicore simulator: fast simulation speed and high scalability.

- The primary goal is to achieve fast simulation speed for multicore simulations in the early design stages. By abstracting the superscalar processor cores in the system and focusing only on the uncore events, I expect to reduce a significant amount of simulation time overhead.

- Simulator's memory space usage is an important aspect when scaling the simulator to support hundreds of cores. The framework consumes a small and constant memory space during simulation regardless of the workload, whereas existing execution-driven simulations do not. Efficient usage of memory space is why such framework is appealing for highly scalable multicore simulator, when hundreds of core counts is considered.

7.1.3 Evaluation methods

Much previous performance modeling works, including simulation and analytical modeling, were validated by comparing the estimated performance of the proposed model to an existing simulator. Making comparison with a real hardware is not practical in many occasions, hence, the reference simulator is assumed to be the golden model for comparison. Comparing the proposed multicore simulator with other state-of-the-art multicore simulators will definitely be a plus, however, it is difficult to make such comparison because of the following reasons. First, to the best of my knowledge, there is no multicore simulator that can simulate hundreds of superscalar processor cores. Second, even if such simulator exits, it is difficult to modify the simulator to make close comparisons with the simulation results. Hence, I plan to instead devise case studies to test the simulator's capability. The case studies will focus on evaluating the relative performance change, and measuring the off-chip bandwidth and the contention on shared resources such as L2 cache and memory controller.

BIBLIOGRAPHY

[1] A. Agarwal and M. Huffman. Blocking: Exploiting spatial locality for trace compaction. In *Proc. ACM SIGMETRICS Conf. Measurement and modeling of computer systems (SIGMETRICS)*, pages 48–57, 1990.

[2] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, Feburary 2002.

[3] T. Ball and J. R. Larus. Efficient path profiling. In *Proc. Int'l Symp. Microarchitecture (MICRO)*, pages 46–57, 1996.

[4] L. Barnes. Performance modeling and analysis for amd's high performance microprocessors, April 2007. Keynote at *Int'l Symp. Performance Analysis of Systems and Software (ISPASS)*.

[5] L. A. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[6] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proc. of the annual conference on USENIX Annual Technical Conference*, pages 41–41, 2005.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT)*, October 2008.

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, May 2011.

[9] P. Bitar. *A critique of trace-driven simulation for shared-memory multiprocessors.* Springer US, 1990.

[10] B. Black, A. S. Huang, M. H. Lipasti, and J. P. Shen. Can trace-driven simulators accurately predict superscalar performance? In *Proc. Int'l Conf. Computer Design (ICCD)*, pages 478–485, October 1996.

[11] Dinero IV cache simulatort. http://pages.cs.wisc.edu/ markhill/DineroIV/.

[12] J. Chame and M. Dubois. Cache inclusion and processor sampling in multiprocessor simulations. In *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 36–47, June 1993.

[13] X. Chen and T. Aamodt. Hybrid analytical modeling of pending cache hits, data prefetching, and mshrs. In *Proc. Int'l Symp. Microarchitecture (MICRO)*, pages 455–465, November 2008.

[14] Sangyeun Cho. I-cache multi-banking and vertical interleaving. In *Proc. ACM Great*

Lakes symposium on VLSI (GLSVLSI), pages 14–19, 2007.

[15] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proc. Int'l Symp. Computer Architecture (ISCA)*, pages 76–87, 2004.

[16] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proc. Int'l Symp. Computer Architecture (ISCA)*, pages 333–344, June 1995.

[17] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual, vol 3a: System programming guide, part1, 2010. http://www.intel.com/products/processor/core2duo/index.htm.

[18] D. Dunning, R. Mooney, P. Stolt, and B. Casper. Tera-scale memory challenges and solutions. *Intel Technology Journal*, 13(4):80–101, 2009.

[19] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. De Bosschere. Statistial simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23(5):26–38, September–October 2005.

[20] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proc. Int'l Symp. Computer Architecture (ISCA)*, pages 373–382, May 1988.

[21] S. J. Eggers, D. R. Keppel, E. J. Koldinger, and H. M. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *Proc. ACM SIGMETRICS Conf. Measurement and modeling of computer systems (SIGMETRICS)*, pages 37–46, May 1990.

[22] M. Ekman and P. Stenström. Enhancing multiprocessor architecture simulation speed using matched-pair comparison. In *Int'l Symp. Performance Analysis of Systems and Software (ISPASS)*, pages 89–99, March 2005.

[23] S. Borkar *et. al.* Platform 2015: Intel processor and platform evolution for the next decade, March 2005. Technology at Intel Magazine.

[24] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems*, 27(2):1–37, May 2009.

[25] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *Proc. Int'l Symp. Computer Architecture (ISCA)*, pages 74–85, June–July 2001.

[26] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn. Interaction cost and shotgun profiling. *IEEE Transactions on Architecture and Code Optimization (TACO)*, 1(3):272–304, September 2004.

[27] K. Ganesan, J. Jo, and L. K. John. Synthesizing memory-level parallelism aware miniature clones for spec cpu2006 and implantbench workloads. In *Int'l Symp. Performance Analysis of Systems and Software (ISPASS)*, pages 33–44, 2010.

[28] D. Genbrugge and L. Eeckhout. Memory data flow modeling in statistical simulation for the efficient exploration of microprocessor design spaces. *IEEE Transactions on Computers (TC)*, 57(1):41–54, Janurary 2008.

[29] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *Proc. Int'l Symp. High-Performance Computer Architecture (HPCA)*, pages 307–318, Janurary 2010.

[30] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. *ACM SIGMETRICS Performance Evaluation Review*, 21(1):146–157, June 1993.

[31] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 4th Ed., 2006.

[32] V. S. Iyengar, L. H. Trevillyan, and P. Bose. Representative traces for processor models with infinite cache. In *Proc. Int'l Symp. High-Performance Computer Architecture (HPCA)*, pages 62–72, February 1996.

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

[34] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. Measuring benchmark similarity using inherent program characteristics. *IEEE Transactions on Computers (TC)*, 55(6):769–782, June 2006.

[35] 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, January 2003.

[36] T. Karkhanis and J. E. Smith. The first-order superscalar processor model. In *Proc. Int'l Symp. Computer Architecture (ISCA)*, pages 338–349, June 2004.

[37] 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.

[38] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, March–April 2005.

[39] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proc. Int'l Symp. Computer Architecture (ISCA)*, pages 81–87, 1981.

[40] J. R. Larus. Efficient program tracing. *Journal Computer*, 26(5):52–61, May 1993.

[41] H. Lee, S. Cho, and B. R. Childers. Cloudcache: Expanding and shrinking private caches. In *Proc. Int'l Symp. High-Performance Computer Architecture (HPCA)*, February 2011.

[42] H. Lee, L. Jin, K. Lee, S. Demetriades, M. Moeng, and S. Cho. Two-phase trace-driven simulation (tpts): A fast multicore processor architecture simulation approach. *Software: Practice and Experience (SPE)*, 40(3):239–258, March 2010.

[43] K. Lee and S. Cho. In-n-out: Reproducing out-of-order superscalar processor behavior from reduced in-order traces. In *Int'l Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 126–135, July 2011.

[44] K. Lee and S. Cho. Accurately modeling superscalar processor performance with reduced trace. *Journal of Parallel and Distributed Computing (JPDC)*, 73(4):509–521, April 2013.

[45] K. Lee, S. Evans, and S. Cho. Accurately approximating superscalar processor performance from traces. In *Int'l Symp. Performance Analysis of Systems and Software (ISPASS)*, pages 238–248, April 2009.

[46] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron. Cmp design space exploration subject to physical constraints. In *Proc. Int'l Symp. High-Performance Computer Architecture (HPCA)*, pages 62–72, February 2006.

[47] D. J. Lilja. *Measuring Computer Performance: A Practitioner's Guide.* Cambridge University Press, 2000.

[48] Gabriel Loh. A time-stamping algorithm for efficient performance estimation of superscalar processors. In *Proc. ACM SIGMETRICS Conf. Measurement and modeling of computer systems (SIGMETRICS)*, pages 72–81, June 2001.

[49] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 190–200, June 2005.

[50] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.

[51] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *Computer Architecture News (CAN)*, September 2005.

[52] P. Michaud, A. Seznec, and S. Jourdan. An exploration of instruction fetch requirement in out-of-order superscalar processors. *Int'l Journal of Parallel Programming*, 29(1):35–58, February 2001.

[53] J. E. Miller, H. Kasture, G. Kurian C. Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proc. Int'l Symp. High-Performance Computer Architecture (HPCA)*, pages 1–12, Janurary 2010.

[54] M. Moeng, S. Cho, and R. Melhem. Scalable multi-cache simulation using gpus. In *Int'l Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 159–167, July 2011.

[55] M. Moudgill, J. Wellman, and J. Moreno. Environment for powerpc microarchitecture exploration. *IEEE Micro*, 19(3), May–June 1999.

[56] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, M. D. Hill, D. A. Wood, S. Huss-Lederman, and J. R. Larus. Wisconsin wind tunnel ii: a fast, portable parallel architecture simulator. In *Proceedings of the Workshop Performance Analysis and its Impact on Design (PAID)*, pages 12–20, June 1997.

[57] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 89–100, June 2007.

[58] D. B. Noonburg and J. P. Shen. A framework for statistical modeling of superscalar processor performance. In *Proc. Int'l Symp. High-Performance Computer Architecture (HPCA)*, pages 298–309, February 1997.

[59] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT)*, pages 15–24, October 2001.

[60] S. Nussbaum and J. E. Smith. Statistical simulation of symmetric multiprocessor systems. In *Proc. Annual Simulation Symp. (SS)*, pages 89–97, April 2002.

[61] D. Ofelt and J. L. Hennessy. Efficient performance prediction for modern microprocessors. In *Proc. ACM SIGMETRICS Conf. Measurement and modeling of computer systems (SIGMETRICS)*, pages 229–239, June 2000.

[62] D. Papworth. Tuning the pentium pro microarchitecture. *IEEE Micro*, 16(2):8–15, April

1996.

[63] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. Marssx86: A full system simulator for x86 cpus. In *Design Automation Conference 2011 (DAC'11)*, 2011.

[64] L. A. Polka, H. Kalyanam, G. Hu, and S. Krishnamoorthy. Package technology to address the memory bandwidth challenge for tera-scale computing. *Intel Technology Journal*, 11(3):197–206, August 2007.

[65] K. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proc. Int'l Symp. Microarchitecture (MICRO)*, pages 24–34, December 1996.

[66] Andre Seznec, Stephan Jourdan, Pascal Sainrat, and Pierre Michaud. Multiple-block ahead branch predictors. In *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 116–127, 1996.

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

[68] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57, October 2002.

[69] A. J. Smith. Cache memories. 14(3):473–530, September 1982.

[70] SPEC. Standard performance evaluation corporation. http://www.specbench.org.

[71] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proc. Int'l High-Performance Computer Architecture (HPCA)*, pages 63–74, February 2007.

[72] TPC. Transactions processing council. http://www.tpc.org.

[73] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29(2):128–170, June 1997.

[74] W.-H. Wang and J.-L. Baer. Efficient trace-driven simulation methods for cache performance analysis. In *Proc. ACM SIGMETRICS Conf. Measurement and modeling of computer systems (SIGMETRICS)*, pages 27–36, May 1990.

[75] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. Simulation sampling with live-points. In *Int'l Symp. Performance Analysis of Systems and Software (ISPASS)*, pages 2–12, March 2006.

[76] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. Simflex: Statistical sampling of computer architecture simulation. In *IEEE Micro special issue on Computer Architecture Simulation*, pages 2–15, July/August 2006.

[77] M. V. Wilkes. The memory wall and the cmos end-point. *ACM SIGARCH Computer Architecture News*, 23(4):4–6, September 1995.

[78] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proc. Int'l Symp. Computer Architecture (ISCA)*, pages 24–36, July 1995.

[79] W. A. Wulf and S. A. Mckee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, March 1995.

[80] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proc. Int'l. Symp. Computer Architecture (ISCA)*, pages 84–95, June 2003.

[81] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. An evaluation of strat-

ified sampling of microarchitecture simulations. In *Proc. Workshop Duplicating, Deconstructing, and Debunking (WDDD) held in conjunction with Int'l. Symp. Computer Architecture (ISCA)*, June 2004.

[82] 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, November 2006.