In-N-Out: Reproducing Out-of-Order Superscalar Processor Behavior from Reduced In-Order Traces

Kiyeon Lee and Sangyeun Cho
Computer Science Department, University of Pittsburgh
Pittsburgh, PA 15260, USA
{lee,cho}@cs.pitt.edu

***Abstract*—Trace-driven simulation is a widely practiced simulation method. However, its use has been typically limited to modeling of in-order processors because of accuracy issues. In this work, we propose and explore In-N-Out, a fast approximate simulation method to reproduce the behavior of an out-of-order superscalar processor with a reduced in-order trace. During trace generation, we use a functional cache simulator to capture interesting processor events such as uncore accesses in the program order. We also collect key information about the executed program. The prepared in-order trace then drives a novel simulation algorithm that models an out-of-order processor. Our experimental results demonstrate that In-N-Out produces reasonably accurate absolute performance (7% difference on average) and fast simulation speeds (115× on average), compared with detailed execution-driven simulation. Moreover, In-N-Out was shown to preserve a processor's dynamic uncore access patterns and predict the relative performance change when the processor's core- or uncore-level parameters are changed.**

***Keywords*—Superscalar out-of-order processor, performance modeling, trace-driven simulation.**

I. Introduction

Various trace-driven simulation methods have been indispensable to computer architects for decades. To run a simulation, a trace of interesting processor events need to be generated prior to simulation. Once the trace has been prepared it is used multiple times (with different simulated system 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 at an early design stage [19]. Unfortunately, the use of trace-driven simulation has been typically limited to modeling relatively simple in-order processors because of accuracy issues; the static nature of the trace poses challenges when modeling a dynamically scheduled out-of-order processor [3], [11]. To model an out-of-order superscalar processor (or simply "superscalar processor" in this paper), it is believed that full tracing and computationally expensive detailed modeling of processor microarchitecture are required [2].

This paper explores *In-N-Out*, a novel fast and storage-efficient approximate simulation method for reproducing a superscalar processor's dynamic execution behavior from a *reduced in-order trace*. Our goal is to provide a practical and effective simulation environment for evaluating a superscalar processor's performance, especially when the focus of the study is on uncore components like the L2 cache and memory controller. Using our environment, one can employ a fast functional simulator like *sim-cache* [1] or *Pin* [13] to generate traces and perform simulations at high speeds. In-N-Out achieves a reasonable absolute performance difference compared with an execution-driven simulation method and can accurately predict the relative performance of the simulated machine when the machine's uncore parameters are changed. We also find that important processor artifacts like data prefetching and miss status handling registers (MSHRs) [10] can be easily incorporated in the In-N-Out framework.

While collecting a trace, In-N-Out monitors the length of the instruction dependency chains, which can have a critical impact on the program execution time. The memory access information and the instruction dependency information are recorded to construct a trace item. During trace simulation, In-N-Out dynamically reconstructs the processor's reorder buffer (ROB) state, honors dependencies between the trace items, and takes into account the length of the dependency chains. Our experimental results demonstrate that In-N-Out, based on simple yet effective ideas, achieves a small CPI difference of 7% and high simulation speeds of 115× on average with the SPEC2K benchmark suite [17], compared with a widely used execution-driven architecture simulator. More importantly, In-N-Out tightly follows the performance changes seen by execution-driven simulation when the uncore configurations, such as L2 cache size and associativity, are changed. The performance change direction is always predicted correctly and the performance change amount is predicted with less than 4% on average.

A. Related work

Previously, researchers proposed analytical performance models to quickly derive the performance of superscalar processors. For example, Karkhanis and Smith [9] proposed a first-order analytical performance model to estimate a superscalar processor's performance. Chen and Aamodt [4] and Eyerman et al. [5] extended the first order model by improving its accuracy and incorporating more processor artifacts. Michaud et al. [14] 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. Like In-N-Out, these proposals employ a target program's in-order trace; however, their goal is to derive the overall program

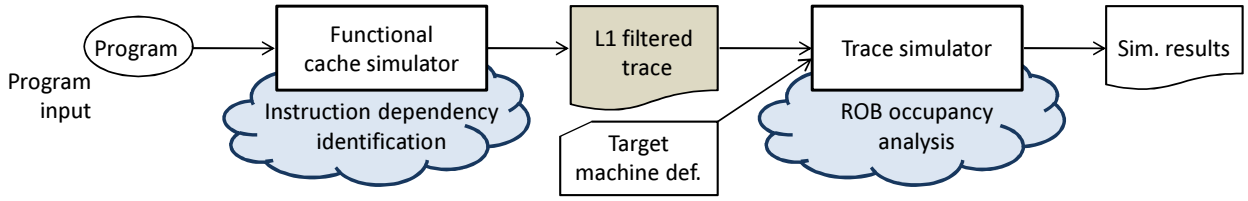

Fig. 1. Overall structure of In-N-Out.

performance from a model constructed from the trace rather than reproduce (or simulate) the dynamic behavior of the processor being modeled. A more comparable recent work by Lee et al. [11] shares the same goal with this work. However, their work assumed that a trace generator is a cycle-accurate architecture simulator so that architecture-dependent timing information can be directly collected during trace generation. We do not make such an assumption and use a simple functional simulator to generate storage-efficient L1 cache filtered traces.

B. Contributions

This paper makes the following contributions:

- We propose and present in detail In-N-Out, a practical trace-driven simulation method effective for modeling superscalar processors (Section II). We discuss the key design issues and quantify their effect. We describe our main algorithms so that future users can reproduce the results of this paper and build their own tools. To the best of our knowledge, our work is the first to study the usage of reduced in-order traces to simulate superscalar processors, and report its quantitative evaluation results.
- We demonstrate that In-N-Out is capable of faithfully replaying how a superscalar processor exercises and is affected by the uncore components (Section IV-B). Our study used a relevant temporal metric—the profile of the distances between two consecutive memory accesses (in cycles). Given that the importance of simulation productivity will only grow with multicore scaling, we believe that our work is the essential first step for developing a very fast and scalable multicore simulator that can model a large number of superscalar processor cores.
- We also present a case study that involves three uncore artifacts: the number of MSHRs, data prefetching, and L2 cache configuration (Section IV-D). Our results reveal that In-N-Out is capable of tightly tracking the performance estimated by an equivalent yet much slower execution-driven simulator. As a result, In-N-Out was able to identify a design point that optimizes the processor design much faster than the execution-driven simulation strategy.

II. In-N-Out

A. Overview

The overall structure of In-N-Out is illustrated in Fig. 1. In-N-Out uses a *functional cache simulator* to quickly generate trace items on L1 data cache misses and write backs. The *L1 filtered trace* is comprised of these trace items. Compared with a full trace where each trace item corresponds to an executed instruction, an L1 filtered trace only includes a small subset of instructions. With no timing information available during trace generation, we focus on the length of the instruction dependency chains in a program by tracking data dependencies between instructions and record the information in the trace. The filtered 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 uncore components. The *trace simulator* runs the simulation algorithm to dynamically reconstruct the ROB state, honor the dependency between trace items, and exploit the recorded length of the dependency chains. Finally, the simulation results are obtained from the trace simulator. In what follows, we will first discuss how we generate L1 filtered traces while monitoring the instruction dependencies. We will then detail the trace simulation algorithm.

B. Preparing L1 filtered traces

1) Identifying instruction data dependency: In trace generation, we measure the length of the dependency chains, by detecting the data dependency between instructions. When an instruction is processed, an instruction sequence number (ISN) is given to the instruction in the program order. When an instruction writes to a register, it labels the output register with its ISN. Later, when an instruction reads data from the same register, the labeled ISN is used to identify the existing dependency. The length of the dependency chain is incremented when a new instruction is included. When two separate dependency chains merge at one instruction, we use the length of the longer 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 number of instructions between the two trace items in the dependency chain. For instructions depending on multiple trace items, we keep the ISNs of the two most recent trace items in the dependency chain. While storing more than two trace items may improve accuracy, we experimentally determined that storing at most two ancestors is sufficient. For a store trace item, we distinguish the data dependency in memory address computation and store operand to correctly handle memory dependencies during trace simulation.

Besides the direct dependency between trace items, an indirect dependency may exist via a "delayed hit". A delayed hit occurs when a memory instruction accesses a cache block that is still in transit from the lower-level cache or the 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 [4]. To take this indirect dependency into account, when an L1 data cache miss occurs, the cache block is labeled with the 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 compared with the ISNs of the memory instruction's ancestor trace items. The two largest ISNs are then written in the memory instruction's output register. Note that a trace item generated by a load can be marked as a dependent of a trace item generated by a store, if it depends on delayed hits created by the store.

2) Estimating instruction execution time: Given a small code fragment of a program, the execution time of the fragment is bounded by the length of the longest dependency chain in the fragment [5], [7], [9], [14]. Based on this observation, we partition the full instruction stream into fixed size "windows" and view the program execution as the sequence of these windows. We then determine the length of the longest dependency chain in each window to estimate its execution time. The total execution time can be derived by simply adding the execution times of all windows. The instruction execution time between two trace items is estimated by $DC\_length_{curr} - DC\_length_{prev}$, which correspond to the measured dependency chain length up to the current trace item and the previous trace item, respectively.

However, there is complication to this approach. Consider a superscalar processor fetching and executing instructions in a basic block. If the processor can fetch and dispatch the entire basic block into the ROB instantly, the length of the longest dependency chain in the basic block can be used to estimate the execution time of the basic block. However, since a processor may need several cycles to fetch and dispatch all the instructions in the basic block, the processor may finish executing instructions in the longest dependency chain before the entire basic block is fetched. Michaud et al. [14] recognized this case and estimated the execution time of a basic block as a function of the instruction fetch time and the length of the dependency chains in the basic block.

In our work, we take a simpler approach. We set the dependency monitoring window (DMW) size small enough to eliminate the case discussed above. We set the DMW size to a multiple of the processor's dispatch-width and measure the length of the longest dependency chain in the DMW. Using this strategy, determining the right DMW size is critical to achieve good estimation accuracy. If the DMW size is too small, we may miss instructions that can actually execute in parallel. However, if it is too large, we may consider more instructions for parallel execution than we should. Moreover, we observed that a single DMW size does not work equally well for all programs. We evaluate the effect of using different DMW sizes on simulation accuracy in Section IV-A.

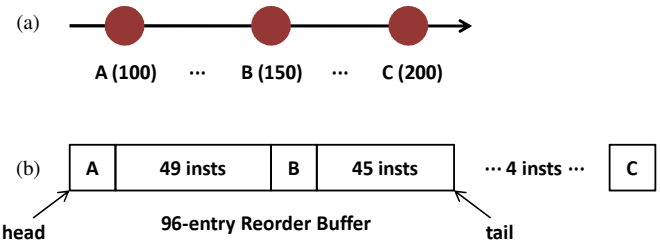

Fig. 2. (a) Three independent L2 cache misses from memory instructions A, B, and C. Inside parentheses is the ISN of the memory instruction. (b) The status of the ROB: instruction C is not in the ROB with A and B.

C. Out-of-Order Trace Simulation

At the heart of out-of-order trace simulation is the ROB occupancy analysis. We will first discuss how the ROB state can stall program execution, and then describe our trace simulation algorithm.

1) ROB Occupancy Analysis: The ROB plays a central role in parallel instruction execution, since only the instructions that are in the ROB can be executed together. For example, with a 96-entry ROB, two memory instructions cannot be simultaneously outstanding if they are 96 or more instructions away from each other [9]. The example in Fig. 2 shows how instructions accessing the memory can stall the execution of the program with limited ROB size.

Suppose all three memory instructions A, B, and C miss in the L2 cache, and A becomes the head of the ROB. Given their ISN, B can be placed in the ROB with A, since the number of instructions between A and B is smaller than the ROB size. However, the number of instructions between A and C is larger than the ROB size. Consequently, C cannot issue a cache access while A is in the ROB. Cache access from B can overlap with the cache access from A. Once the ROB is full and there are no more instructions to issue, the program execution stalls until A commits. After A commits and exits the ROB, the program execution resumes. The issued instructions between A and B also commit at the processor's commit rate, which allows the instructions behind the tail of the ROB, including C, to enter the ROB until ROB is not full.

Based on this observation, we reconstruct the ROB during trace simulation. In trace generation, when a trace item is generated on L1 cache miss, the ISN of the L1 cache miss is recorded in the trace item. During simulation, we use the ISN of the trace item to determine which trace items can be dispatched to the ROB. We allow all L2 cache accesses (trace items) with no dependency stalls in the ROB to issue, and stop fetching trace items when the fetched trace item is farther from the ROB head by at least the ROB size.

2) Implementing out-of-order trace simulation: Table I lists the notations used in the rest of this section. In trace simulation, we employ two lists to implement our simulation algorithm: *rob-list* and *issue-list*. *rob-list* links trace items in program order to reconstruct the ROB state during trace simulation. Trace items are inserted into *rob-list* if the difference

TABLE I
NOTATIONS USED IN SECTION II-C2.

DC	The instruction dependency chain
rob-list	The list used to link trace items with respect to trace item's ISN
issue-list	The list used to link trace items with respect to trace item's *ready_time*
rob_head	The trace item in the head of *rob-list*
issue_head	The trace item in the head of *issue-list*
sim_time	The current clock cycle time
ready_time	The time when a trace item is ready to be processed
return_time	The time when the trace item's cache access returns
trace_process_time	The time to process *issue_head*
rob_head_commit_time	The time to remove *rob_head* from *rob-list*

```
1: while (1) do
2:    sim_time++;
3:    if (sim_time == rob_head_commit_time) then
4:       Commit_Trace_Items();
5:       Update_ROB();
6:       update rob_head_commit_time for the new rob_head
7:    end if
8:    if (sim_time == trace_process_time) then
9:       Process_Trace_Items();
10:   end if
11:   if (no more trace items left in the trace file) then
12:      break; /* END OF TRACE SIMULATION */
13:   end if
14: end while
```

Fig. 3. Pseudo-code of our trace simulation algorithm.

between the trace item's ISN and *rob_head*'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, we determine 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 *rob_head* is removed. This reflects 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 execution resumes after the memory instruction commits and new instructions are dispatched into the ROB.

Fig. 3 presents the high-level pseudo-code of our trace simulation algorithm to model the superscalar processor with the baseline configuration described in Section III-A. The key steps are described below.

Commit_Trace_Items. The time to remove *rob_head* from *rob-list* is indicated by *rob_head_commit_time*. Depending on the commit-width, more than one trace item can be removed from *rob-list*. If *rob_head* was generated by a store instruction, we make a write access to L2 cache before we remove *rob_head*. After *rob_head* is removed, the next trace item in *rob-list* becomes the new *rob_head*.

Update_ROB. After committing trace items, we attempt to insert new trace items in *rob-list*. Since multiple trace items are inserted in *rob-list* simultaneously, we estimate when the new trace item will actually be dispatched in the ROB. Assuming that the ROB is full at *rob_head_commit_time*, we define the $dispatch\text{-}time$ as,

$$dispatch\text{-}time = rob\_head\_commit\_time + \left\lceil \frac{ISN_{new\_trace\_item}}{dispatch\text{-}width} \right\rceil - \left\lceil \frac{ISN_{commit\_rob\_head} + ROB\ size}{dispatch\text{-}width} \right\rceil$$

where $ISN_{new\_trace\_item}$ represents the ISN of the new trace item and $ISN_{commit\_rob\_head}$ is the ISN of the trace item that was committed at *sim_time*. At the start of the trace simulation, when *rob-list* is constructed for the first time, the $dispatch\text{-}time$ of the trace items is $sim\_time + \left\lceil \frac{ISN_{new\_trace\_item}}{\text{dispatch-width}} \right\rceil$. If the inserted trace item has no dependency to address, the trace item's *ready_time* is set to "dispatch-time + 1". If the trace item depends on a preceding trace item in *rob-list*, the *ready_time* is set to $MAX(dispatch\text{-}time+1, dependency\text{-}resolve\text{-}time)$, where $dependency\text{-}resolve\text{-}time$ is the parent trace item's *return_time* plus the number of instructions between the parent trace item and the new trace item in the dependency chain.

Update rob_head_commit_time. After updating *rob-list*, *rob_head_commit_time* is set to $MAX(sim\_time + inst\_exec\_time, rob\_head's\ return\_time + 1)$ for the new *rob_head*, where $inst\_exec\_time$ is the time required to issue and commit instructions between the previous and current *rob_head*. We estimate $inst\_exec\_time$ as,

$$inst\_exec\_time = MAX(\left\lceil \frac{ISN_{rob\_head}}{commit\text{-}width} \right\rceil - \left\lceil \frac{ISN_{prev\_rob\_head}}{commit\text{-}width} \right\rceil, recorded\ DC\ length\ in\ rob\_head)$$

where ISN_{rob_head} and $ISN_{prev\_rob\_head}$ are the ISN of current and previous *rob_head*, respectively.

Process_Trace_Items. The time to process *issue_head* is indicated by *trace_process_time*. If *issue_head* was generated by a load instruction, we first make a read access to the L2 cache and then search for dependent trace items in *rob-list*. If a dependent trace item is identified, we set its *ready_time* to $dependency\text{-}resolve\text{-}time$ as described above. If *issue_head* was generated by a store instruction, we set *issue_head*'s *return_time* to *sim_time* and perform a write access when it is removed from *rob-list*; i.e., when it commits. Memory dependency is examined when we search *rob-list* to find ready trace items after a cache access. If there is a store trace item with unresolved memory address dependency in *rob-list*, all load trace items behind the store trace item are not inserted in *issue-list*.

TABLE II
BASELINE MACHINE MODEL.

Dispatch/issue/commit width	4
Reorder buffer	96 entries
Load/Store queue	96 entries
Integer ALUs ($\forall$ FU lat. = 1)	4
Floating point ALUs ($\forall$ FU lat. = 1)	2
L1 i- & d-cache (perfect i-cache)	1 cycle, 32KB, 8-way, 64B line size, LRU (Perfect ITLB and DTLB)
L2 cache (unified)	12 cycles, 2MB, 8-way, 64B line size, LRU
Main memory latency	200 cycles
Branch predictor	Perfect

3) Incorporating a data prefetcher and MSHRs: Important processor artifacts such as data prefetching and MSHRs can be easily added into our simulation algorithm.

Modeling the 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.

Modeling the MSHRs. We assume that an MSHR can hold the miss and delayed hits to a cache block. Since the number of outstanding cache accesses is now limited by the available MSHRs, *issue_head* or *rob_head* cannot issue a cache access if there is no free MSHR.

III. EXPERIMENTAL SETUP

A. Machine model

Table II lists our baseline model's superscalar processor configuration, which resembles the Intel Core 2 Duo processor [6]. In Section IV, like previous proposals [4], [11], we assume perfect branch prediction and instruction caching as our focus is on the out-of-order issue and the data memory hierarchy of the superscalar processor architecture. We will separately discuss how to incorporate branch prediction and instruction caching within In-N-Out in Section IV-E.

We use two different machine models in experiments, "baseline" and "combined." The baseline model assumes infinite MSHRs and no data prefetching. The more realistic combined model incorporates data prefetching and MSHR. For prefetching, we implemented two sequential prefetching techniques, prefetch-on-miss and tagged prefetch [16] and stream-based prefetching [18].

The baseline and combined machine configurations will be simulated with `sim-outorder` [1] ("`esim`") and our In-N-Out trace-driven simulator ("`tsim`"). `sim-outorder` has been largely used as a counterpart when verifying a new simulation method or an analytical model for superscalar processors [4], [5], [9], [11], [15], [20]. We extended `sim-outorder` with data prefetching and MSHRs. `tsim` implements the algorithm described in Section II-C2. To drive `tsim`, we adapt `sim-cache` functional simulator [1] to generate traces.

TABLE III
INPUTS USED FOR THE 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

B. Benchmarks

We use all benchmarks from the SPEC2K suite. SPEC2K programs are valid benchmarks for our work, since it offers a variety of core and uncore resource usage patterns. When we show how closely `tsim` reproduces the superscalar processor behavior, we use a selected set of benchmarks rather than all benchmarks for more intuitive presentation: *mcf*, *gcc*, *gzip*, *twolf* (integer), *fma3d*, *applu*, *equake*, and *mesa* (floating point). Our selections are drawn from the eight clusters present in the SPEC2K suite, formed with principal component analysis (PCA) and K-means clustering techniques applied to key microarchitecture-independent characteristics of the benchmark programs [8]. Table III shows the inputs used for the benchmarks. Programs were compiled using the Compaq Alpha C compiler (V5.9) with the `-O3` optimization flag. For each simulation, we skip the initialization phase of the benchmark [15], warm up caches in the next 100M instructions, and simulate the next 1B instructions.

C. Metrics

Throughout Section IV we use *CPI error* and *relative CPI change* as the main metrics. CPI error is defined as $(CPI_{tsim} - CPI_{esim})/CPI_{esim}$, where CPI_{tsim} and CPI_{esim} are the CPI obtained with `tsim` and `esim`, respectively. The CPI error represents the percentage of cycle count difference we obtain with `tsim` compared with `esim`. A negative CPI error suggests that the simulated cycle count with `tsim` is smaller. The average CPI error is obtained by taking the arithmetic mean of the absolute CPI errors, as similarly defined in [4]. Relative CPI change is used to measure the performance change of a machine configuration relative to the performance of another (baseline) configuration. It is defined as $(CPI_{conf2} - CPI_{conf1})/CPI_{conf1}$, where CPI_{conf1} represents the CPI of a base configuration and CPI_{conf2} represents the CPI of a revised configuration [12]. A negative relative CPI change suggests that the simulated cycle count with CPI_{conf2} is smaller than CPI_{conf1}, i.e., performance improved with $conf2$ relative to $conf1$. In addition, we use *relative CPI difference* to compare the performance change amount of `esim` and `tsim`. Relative CPI difference

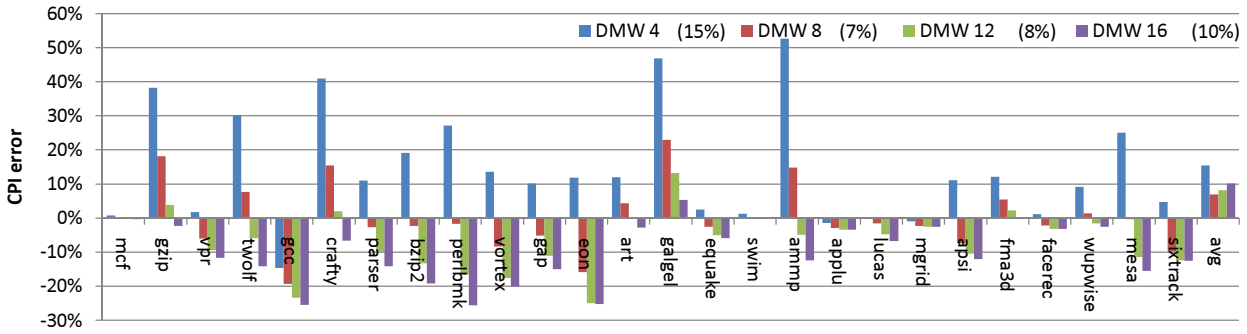

Fig. 4. The CPI error with different DMW sizes. Inside parenthesis is the average CPI error of each DMW size.

is defined as $|rel\_cpi\_chg_{esim} - rel\_cpi\_chg_{tsim}|$, where $rel\_cpi\_chg_{esim}$ and $rel\_cpi\_chg_{tsim}$ are the relative CPI change reported by `esim` and `tsim`, respectively.

IV. Quantitative Evaluation Results

In this section, we comprehensively evaluate In-N-Out. Note that only a single trace file is used for each benchmark to run all experiments.

A. Model accuracy

Baseline configuration. We start evaluating `tsim` with the baseline configuration. Fig. 4 shows the effect of using different DMW sizes on trace simulation accuracy. The size of the DMW has a large effect on the accuracy except for memory-intensive benchmarks, such as *mcf* and *swim*. For all benchmarks, `tsim` showed smaller CPI when larger DMW was used in trace generation.

When DMW size was 4, it was not big enough to bring in instructions in independent dependency chains, whereas `esim` could contain up to 96 instructions in its window (ROB) and execute instructions from independent dependency chains in parallel. `tsim` showed larger CPI than `esim` for 20 (out of 26) benchmarks. When DMW size was increased to 8, the CPI error decreased significantly for many benchmarks, since we were able to monitor more independent dependency chains together.

gzip, *galgel*, and *fma3d* showed steady improvement in trace simulation accuracy with larger DMW. However, many benchmarks showed larger CPI error, smaller CPI compared with *esim*, when DMW size was larger than 8. This is because we assume all instructions in the DMW are fetched instantly, whereas the instructions that we monitor in the DMW are not fetched at once in `esim`. Hence, the estimated instruction execution time in trace generation becomes smaller than the time measured with `esim`. The discrepancy becomes larger when larger DMW is used in trace generation.

The results show that our method provides relatively small CPI error. The average CPI error was 7% when DMW size was 8 and 10% when DMW size was 16. There are benchmarks that show large CPI errors regardless of the DMW size, such as *gcc* and *eon*. We suspect that there may be other important resource conflicts to consider besides the ROB size. Our result suggests that adjusting the DMW size adaptively may increase the simulation accuracy. We leave reducing the CPI errors as our future work. In the rest of this section, we will fix the DMW size to 8 for all benchmarks.

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 many different configurations. In the following results, we will present `tsim`'s capability to closely predict the relative CPI change seen by `esim`.

Combined configuration. Let us turn our attention to how `tsim` performs with regard to data prefetching and MSHRs. We use a relative metric here to compare the configurations with and without these artifacts. Fig. 5(a) presents our results, comparing the relative CPI change with `esim` and `tsim` when different prefetching techniques are employed. Among the eight selected benchmarks, *equake* obtained the largest benefit from data prefetching. *applu* shows good performance improvement with tagged prefetching. Overall, `tsim` closely follows the performance trend revealed by `esim`.

Fig. 5(b) compares the relative CPI change with finite MSHRs. Again, the result demonstrates that `tsim` closely reproduces the increase in cycle count. The largest relative CPI change was shown by *fma3d* with 4 MSHRs—325% with `esim` and 309% with `tsim`.

Finally, the CPI error with the combined configuration is presented in Fig. 6. Comparing Fig. 4 and Fig. 6, we observe that `tsim` maintains the average CPI error of the baseline configuration.

B. Impact of uncore components

Until now, our evaluation of `tsim` has focused on comparing the CPI of `esim` and `tsim` by measuring the CPI error or the relative CPI change. In this subsection we will address two important questions about how well `tsim` captures the interaction of a processor core and its uncore components: (1) Does `tsim` faithfully reproduce how a processor core exercises uncore resources? and (2) Can `tsim` 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

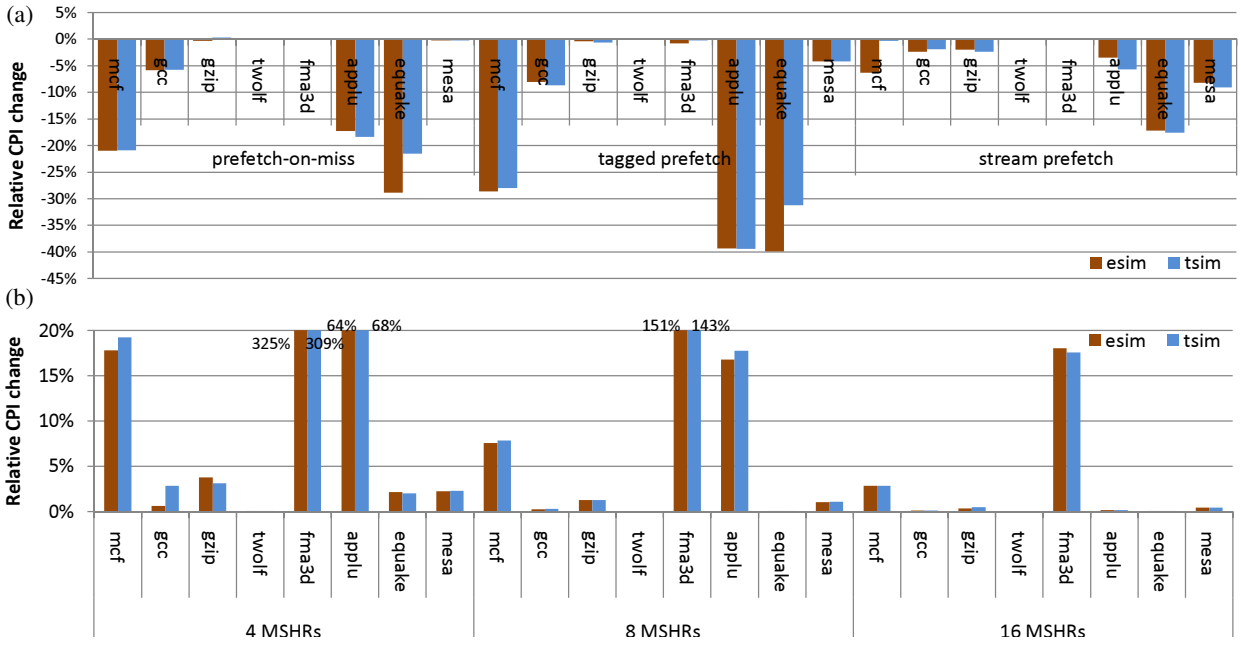

Fig. 5. (a) The relative CPI change when different prefetching techniques are used, compared with no prefetching. For stream prefetching, the prefetcher's prefetch distance is 64 and the prefetch degree is 4, and we track 32 different streams. (b) The relative CPI change when 4, 8, and 16 MSHRs are used, compared with unlimited MSHRs.

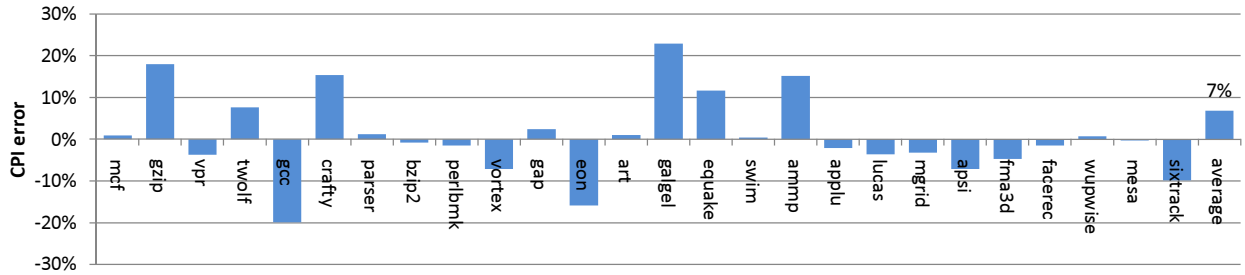

Fig. 6. The CPI error with the combined configuration.

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, we build histograms of the distance (in cycles) between two consecutive memory accesses (from L2 cache misses, write backs, or L2 data prefetching) over the program execution with `esim` and `tsim`. Our intuition is that if `tsim` preserves the memory access patterns of `esim`, 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 memory accesses. The value in a bin represents the frequency of distances that fall into the corresponding range. Fig. 7(a) and (b) depict the histograms of the first interval of *mgrid* and *gcc*. Both plots show that `tsim` preserves the temporal memory access patterns of the programs fairly well. We define a metric to compare `esim` and `tsim` with a single number as follows,

$$Similarity = \frac{\sum_{i=0}^{n} \text{MIN}(bin\_esim_i, bin\_tsim_i)}{\sum_{i=0}^{n} bin\_esim_i}$$

where i is the bin index and bin_esim_i and bin_tsim_i are the frequency value in i_{th} bin collected by `esim` and `tsim`, respectively. The $MIN(bin\_esim_i, bin\_tsim_i)$ returns the common population between `esim` and `tsim` in i_{th} bin. High similarity value implies `tsim`'s ability to preserve the memory access pattern of `esim`. If the similarity is 1, it suggests that the frequency of the collected distances between memory accesses in the two simulators is identical. Table IV presents the computed average $Similarity$ over all intervals for all SPEC2K benchmarks. All, except one, showed 90% or higher similarity (18 was higher than 95%).

To address the second question, Fig. 8 compares the relative CPI change obtained with `esim` and `tsim` when six important uncore parameters are changed. Changing an uncore parameter makes the memory access latencies seen by the processor core different. The average relative CPI difference is reported for all SPEC2K benchmarks. A short bar simply means a

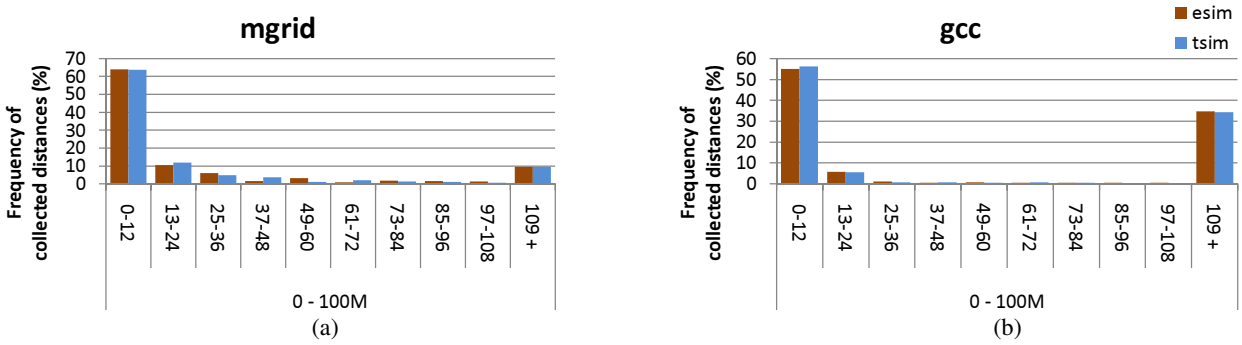

Fig. 7. The histogram of collected distances between two consecutive memory accesses in `esim` and `tsim` when executing (a) *mgrid* and (b) *gcc*. The x-axis represents the bins used to collect the distances and the y-axis represents the frequency of collected distances in the first interval of the program execution. We only show the first interval as it is representative.

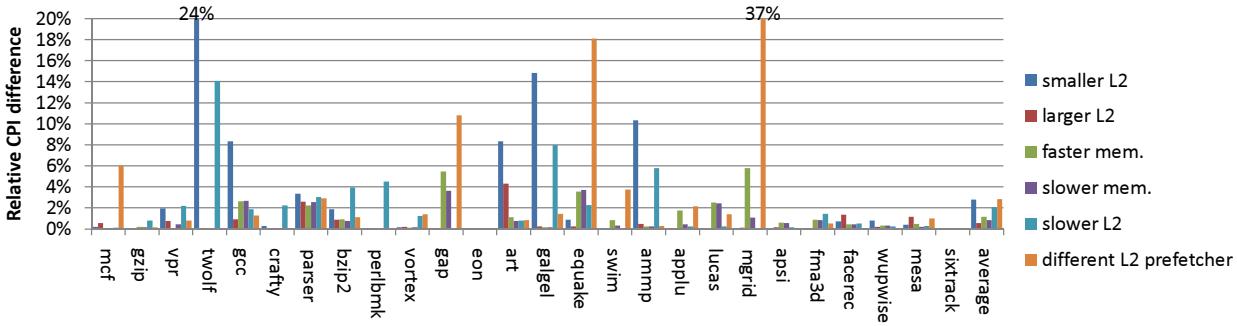

Fig. 8. Comparing the reported relative CPI change by `esim` and `tsim`. Six new configurations are used with a change in uncore parameters. In the first and second configuration, the L2 cache size is 1MB and 4MB instead of 2MB ("smaller L2 cache" and "larger L2 cache"). In the third and fourth configuration, the memory access latency is 100 and 300 cycles instead of 200 cycles ("faster memory" and "slower memory"). In the fifth configuration, the L2 hit latency is 20 cycles instead of 12 cycles ("slower L2 cache"). Lastly, in the sixth configuration, we use a stream prefetcher instead of a tagged prefetcher in L2 cache ("different L2 prefetcher").

TABLE IV
THE SIMILARITY IN MEMORY ACCESS PATTERNS BETWEEN `esim` AND `tsim` (SHOWN IN PERCENTAGE).

Similarity	Benchmark (similarity)
$< 90\%$	*mgrid* (84%)
$\geq 90\%$	*gzip* (91%) *art*, *equake* (92%), *lucas* (93%) *wupwise*, *parser*, *swim* (94%)
	fma3d, *gap*, *ammp*, *applu* (96%) *bzip2*, *vpr* (97%), *perl*, *galgel* (98%)
	crafty, *mesa*, *mcf*, *gcc*, *apsi* *vortex*, *facerec* (99%)
	sixtrack, *eon*, *twolf* (100%)

small relative CPI difference between `esim` and `tsim`. For example, when 2MB L2 cache is changed to 1MB L2 cache, *gcc* experiences a relative CPI change of 18% with `esim` and 26% with `tsim`. The relative CPI difference of the two is 8%, which is shown on the *gcc*'s leftmost bar in Fig. 8. Note that the performance change directions predicted by `esim` and `tsim` always agreed. The largest relative CPI difference (37%) was shown by *mgrid* when the tagged prefetcher is replaced with a stream prefetcher. Overall, the relative CPI differences were very small—the arithmetic mean of the relative CPI difference was under 4% for all six new configurations.

C. Simulation speed

The biggest advantage of using `tsim` over `esim` is its very fast simulation speed. We measured the absolute simulation speed of `tsim` and speedups over `esim` with the combined configuration on a 2.26GHz Xeon-based Linux box with an 8GB main memory. The observed absolute simulation speeds range from 10 MIPS (mcf) to 949 MIPS (sixtrack) and their average is 156 MIPS (geometric mean). The observed simulation speedups range from 13× (*art*) to 684× (*eon*) and their average (geometric mean) is 115×. Note that this is the actual simulation speedup without including the time spent for fast-forwarding in `esim`. When we take the fast-forwarding period of the execution-driven simulation into account, the average simulation speedup was 180×.

The actual trace file size used was 11 (*sixtrack*) to 4,168 (*mcf*) in bytes per 1,000 simulated instructions. We can further reduce the size 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.

D. Case study

Our evaluation results so far strongly suggest that In-N-Out offers adequate performance prediction accuracy for studies comparing different machine configurations. Moreover, In-N-

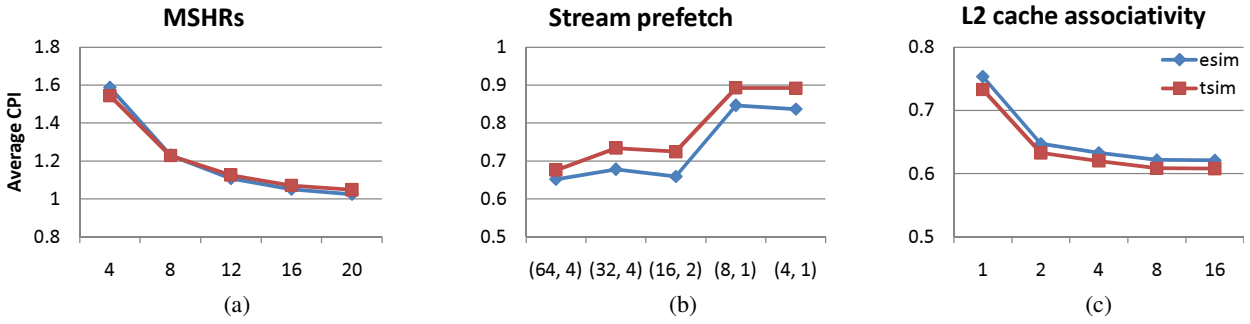

Fig. 9. Comparing the trend in performance (average CPI) change of the superscalar processor with `esim` and `tsim`. The effects of MSHRs, L2 data prefetching, and L2 cache configuration on performance are studied using the two simulators. The following changes have been made on the combined configuration: (a) 5 different MSHRs: 4, 8, 12, 16, and 20 MSHRs. (b) 5 different stream prefetcher configurations (prefetch distance, prefetch degree). (c) 5 different L2 cache associativity: 1, 2, 4, 8, and 16. For each study, we compared the top eight benchmarks that showed the largest performance change amount when observed with `esim`.

Out was shown to successfully reconstruct a superscalar processor's dynamic uncore access behavior. To further show the effectiveness of In-N-Out, we design and perform a case study which involves MSHRs, a stream prefetcher, and L2 cache. In this study, we selected three different sets of programs for each experiment. Each set has eight programs that are most sensitive to the studied parameter.

When the number of MSHRs increases, the CPI decreases because more memory accesses can be outstanding simultaneously. Both `esim` and `tsim` reported the largest decrease in CPI when the number of MSHRs increased from 4 to 8 as shown in Fig. 9(a). The CPI becomes stable when more MSHRs are provided. The close CPI change is a result of `tsim`'s good reproduction of `esim`'s temporal memory access behavior.

In the stream prefetcher, the larger (smaller) prefetch distance and prefetch degree makes the prefetcher more aggressive (conservative) when making prefetching decisions [18]. In general, the CPI increases if the prefetcher becomes more conservative. In Fig. 9(b), both `esim` and `tsim` reported the largest CPI increase when the performance distance and degree were changed from (16, 2) to (8, 1). Finally, Fig. 9(c) shows that when the L2 cache associativity is increased, both `esim` and `tsim` reported the largest decrease in CPI when the L2 cache associativity changed from 1-way to 2-way associativity. Different stream prefetcher configuration or different set-associativity has an effect on CPI by changing the cache miss rate. The close CPI trend obtained with `esim` and `tsim` shows that `tsim` correctly follows how the core responds to different uncore access latencies (e.g., cache misses). The results shown in our case study suggest that `tsim` can be effectively used in the place of `esim` to study the relatively fine-grain configuration changes.

E. Effect of i-caching and branch prediction

There are two reasons why we discuss the issues with instruction caching and branch prediction separately in this paper. First, the main target of the proposed In-N-Out approach is not to study "in-core" parameters like L1 instruction cache and branch predictor, but rather to abstract a superscalar processor core. Second, our goal is to validate In-N-Out by focusing on two aspects of a superscalar processor—handling dynamic out-of-order issue and reproducing (parallel) memory access behavior. Nevertheless, we briefly describe below strategies one can use to model the effect of instruction caching and branch prediction.

To model the instruction caching effect, during trace generation, we can generate trace items on L1 instruction cache misses. The penalties from instruction cache misses can be accounted for at the simulation time by stalling simulation when an instruction cache miss trace item is encountered and if there are no trace items left in the ROB. Our experiments reveal that with this simple strategy In-N-Out can accurately predict the increased clock cycles due to instruction cache misses. In fact, compared with a perfect instruction cache, the CPI increase with a realistic 32KB instruction cache (similar to the data cache in Table II) was 1% on average in both `esim` and `tsim`. Hence, our evaluation of In-N-Out is not affected by the perfect instruction cache used in the experiments.

To account for the effect of branch prediction, we can employ a branch predictor in the trace generator. This task is more involving than modeling the instruction caching effect because branch misprediction penalty depends on microarchitectural parameters like the number of pipeline stages. In our preliminary investigation, we use `sim-outorder` with a combined branch predictor to generate and annotate traces. We record the timing information, including the branch misprediction penalties, in trace items and generate trace items on both correctly and incorrectly predicted control paths. The information is then utilized in simulation time. Using our representative benchmarks we found that branch handling overheads in `esim` and `tsim` agree across the examined programs and the CPI increase due to branch mispredictions was 10% in `esim` and 9% in `tsim` on average. We also found a majority of branch instructions' dynamic behavior (e.g., the number of correct or incorrect predictions) to be fairly stable and robust even if memory access latency changes.

In summary, our empirical results reveal that we can reasonably accurately model the effect of instruction caching and branch prediction within the In-N-Out framework. Incorporat-

ing these two "in-core" artifacts do not affect the effectiveness of our main algorithm.

V. Conclusions

This paper introduced In-N-Out, a novel trace-driven simulation strategy to evaluate out-of-order superscalar processor performance with reduced in-order traces. By using filtered traces instead of full traces, In-N-Out requires less storage space to prepare traces and reduces the simulation time. We 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. We also showed that it 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 a simulation speedup of 115× on average when running the SPEC2K benchmark suite. We conclude that In-N-Out provides a very practical and versatile framework for superscalar processor performance evaluation.

Acknowledgement

This work was supported in part by the US NSF grants: CCF-1064976, CNS-1059202, and CCF-0702236; and by the Central Research Development Fund (CRDF) of the University of Pittsburgh.

References

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

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

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

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

[5] 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.

[6] Intel Corporation. "Intel 64 and IA-32 Architectures Software Developer's Manual, vol 3A: System Programming Guide, Part1," http://www.intel.com/products/processor/core2duo/index.htm, 2010.

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

[8] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John, "Measuring Benchmark Similarity Using Inherent Program Characteristics," *IEEE TC*, 55(6):769–782, Jun. 2006.

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

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

[11] K. Lee, S. Evans, and S. Cho, "Accurately Approximating Superscalar Processor Performance from Traces," *Int'l Symp. Performance Analysis of Systems and Software (ISPASS)*, pp. 238–248, Apr. 2009.

[12] D. J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*, Cambridge University Press, 200.

[13] C. K. Luk et al. "Pin: building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pp. 190–200, Jun. 2005.

[14] P. Michaud et al. "An exploration of instruction fetch requirement in out-of-order superscalar processors," *Int'l Journal of Parallel Programming*, 29(1):35–58, Feb. 2001.

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

[16] A. J. Smith. "Cache memories," *ACM Computing Surveys*, 14(3):473-530 Sep. 1982.

[17] SPEC. "Standard Performance Evaluation Corporation," `http://www.specbench.org`.

[18] S. Srinath et al. "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," *Proc. Int'l High-Performance Computer Architecture (HPCA)*, pp. 63–74, Feb. 2007.

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

[20] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," *Proc. Int'l. Symp. Computer Architecture (ISCA)*, pp. 84–95, Jun. 2003.