Author's personal copy

J. Parallel Distrib. Comput. 73 (2013) 509–521

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Accurately modeling superscalar processor performance with reduced trace

Kiyeon Lee [*,1], Sangyeun Cho

Computer Science Department, University of Pittsburgh, Pittsburgh, PA 15260, USA

ARTICLE INFO

Article history:
Received 24 March 2012
Received in revised form
28 November 2012
Accepted 5 December 2012
Available online 20 December 2012

Keywords:
Simulation methodology
Trace-driven simulation
Out-of-order superscalar processor

ABSTRACT

Trace-driven simulation of out-of-order superscalar processors is far from straightforward. The dynamic nature of out-of-order superscalar processors combined with the static nature of traces can lead to large inaccuracies in the results when the traces contain only a subset of executed instructions for trace reduction. In this paper, we describe and comprehensively evaluate the *pairwise dependent cache miss model* (PDCM), a framework for fast and accurate trace-driven simulation of out-of-order superscalar processors. The model determines 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. Our experimental results demonstrate that a PDCM-based simulator produces highly accurate simulation results (less than 3% error) with fast simulation speeds (62.5× on average) compared with an execution-driven simulator. Moreover, we observed that the proposed simulation method is capable of preserving a processor's dynamic off-core memory access behavior and accurately predicting the relative performance change when a processor's low-level memory hierarchy parameters are changed.

© 2012 Elsevier Inc. All rights reserved.

1. Introduction

Simulation is an important tool for computer architects [30]. It enables one to quickly analyze the behavior of a complex system and to evaluate subtle design trade-offs in a controlled experimental environment. However, its use is limited to situations where it is both reasonably accurate and fast. Trace-driven simulation is a widely used simulation method when traces are prepared and fast simulation is required especially in an early design stage.

Trace-driven simulation's increased speed results from replacing the detailed functional execution of a benchmark with a highly representative trace of a program execution. The trace may capture every executed instruction of a program, or it may contain the information of certain events, such as L2 cache accesses [27]. Trace-driven simulation with a full instruction trace is a widely used method to precisely model the performance of an out-of-order superscalar processor[2] [3]. However, large disk space is required to store a full trace file. Moreover, simulating the full instruction trace slows down the trace simulation speed. Alternatively, one may employ a *filtered trace* instead of a full trace to model superscalar processor performance. Compared with a full trace, a filtered trace can achieve significantly faster simulation speed and smaller storage space, while maintaining the simulation accuracy close to execution-driven simulation [19].

Filtered trace-driven simulation works well for in-order single-issue cores. For example, during the trace generation phase, one might record the type and address of every memory operation as well as the number of instructions executed and the number of cycles elapsed since the last memory operation. This filtered trace only includes memory accesses and summarizes the instructions executed between operations. Because the core executes instructions in order and blocks while waiting for a memory access, the filtered trace would be the same regardless of the memory configuration. Thus one could simulate many different memory hierarchy configurations using the same trace.

However, many issues arise when the same "block-and-go" scheme is applied to superscalar processors. A superscalar processor does not necessarily block during a memory access. Modern processors often execute instructions during the memory access to hide its latency cost. Previously, we examined three strategies to approximate the impact of a long-latency memory access on superscalar processor performance with filtered traces, based on *isolated cache miss model*, *independent cache miss model*, and *pairwise dependent cache miss model* [18]. The isolated cache miss model is capable of accurately quantifying the impact of

* Correspondence to: San #24 Nongseo-Dong, Giheung-Gu, Yongin-City, Gyeonggi-Do, South Korea.

E-mail addresses: kiyeon00.lee@samsung.com (K. Lee), cho@cs.pitt.edu (S. Cho).

1 Currently working at the System LSI Division of Samsung Electronics.

2 In this paper, we use the terms "out-of-order superscalar processor" and "superscalar processor" interchangeably.

0743-7315/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2012.12.002

Author's personal copy

an "isolated" L1 cache miss. The model compares the number of elapsed cycles after processing an L1 cache miss as L2 cache hit and L2 cache miss. However, the proposed approach cannot accurately compute the impact of overlapping L1 cache misses. To accurately model the impact of both isolated and overlapping L1 cache misses, we proposed the independent cache miss model. The model determines when an L1 cache miss trace item can be processed by dynamically reconstructing a processor's reorder buffer (ROB) state during simulation. Lastly, the pairwise dependent cache miss model improves the independent cache miss model by honoring the dependency between L1 cache misses. Among the three strategies, the pairwise dependent cache miss model (PDCM) was shown to achieve the highest simulation accuracy. While this previous work formed a strong basis for fast and accurate trace-driven simulation of superscalar processors, it used a fairly simple processor model without some key processor artifacts like branch prediction, instruction caching, prefetching, and miss status handling registers (MSHRs) [16]. Other related work also did not study all artifacts together [5,7,14,22].

This work improves the accuracy of PDCM and extends PDCM with techniques to incorporate previously ignored processor artifacts. The impact of these artifacts on program execution times was measured to be large—up to 61% (prefetching), 48% (branch prediction), and 329% (MSHRs). We explore and evaluate individual techniques to model the artifacts and assess their combined effectiveness based on a full-fledged processor configuration. Our results demonstrate that the proposed techniques are effective, leading to very small errors of less than 3% and a fairly high simulation speedup of 62.5× on average, compared with detailed execution-driven simulation.

Furthermore, we examined whether PDCM preserves the temporal off-core memory access patterns of a program. This property is important for studies targeting the usage and contention behavior of system-wide resources such as shared L2 cache, on-chip interconnect, and memory controller. We show experimentally that a majority of program execution intervals (typically over 90% of all intervals) preserve temporal off-core memory access patterns. The result suggests that our approach could potentially be used in studies focusing on multicore system resources rather than processor core parameters such as L1 cache configuration and issue width.

Finally, PDCM is shown to robustly predict the relative performance of a new machine configuration once the result for the baseline machine configuration has been obtained. The performance change direction is always predicted correctly and the performance change amount is predicted with a small error of less than 4% on average.

In summary, PDCM accurately models the performance of a realistic superscalar processor by first replacing the processor core simulation with a filtered trace and then simulating the accesses to the uncore components using the filtered trace. Using PDCM, one can quickly evaluate various optimization techniques on uncore components or study the relative performance by changing the uncore configurations.

In the remainder of this paper, Section 2 first describes our superscalar processor model. Section 3 describes PDCM in detail, and Section 4 describes the experimental setup. We quantitatively evaluate PDCM in Section 5. Section 6 summarizes related work and we draw our conclusions in Section 7.

2. Machine model

Our machine model is a superscalar processor system with two levels of cache memory and a main memory, as shown in Fig. 1. The superscalar processor core model we use is sketched inside the dotted box. It has a front-end "fetch pipe" that fetches and buffers

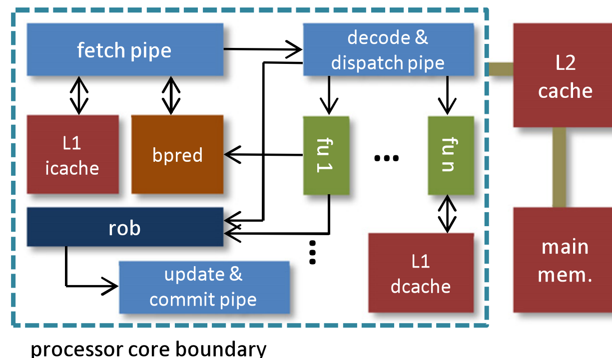


Fig. 1. Machine model having a superscalar processor core, L2 cache, and main memory.

instructions for further processing. Once fetched, instructions are decoded and dispatched to various functional units such as an ALU, branch unit, or data memory access unit. They may be temporarily stored in buffers (or *reservation stations*) associated with a specific functional unit until the unit becomes available or input operands arrive. When an instruction is dispatched, a new entry is allocated in the reorder buffer (ROB) so that the "update and commit pipe" can change the architectural state properly in the program order as instructions are committed in the presence of special events such as exceptions, branch mis-predictions, and cache misses. When a memory access misses in an L1 cache, it accesses an off-core unified L2 cache. If the access misses in the L2 cache, it will access the main memory.

Our processor pipeline design resembles that of MIPS R10k processor [29] and Intel Core [11]. More general description of superscalar processor design and operation can be found in Johnson [12] and Shen and Lipasti [23].

3. Pairwise Dependent Cache Miss Model (PDCM)

A superscalar processor dynamically selects and executes multiple instructions in parallel. As a result, more than one cache miss can be outstanding. 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. With a 96-entry reorder buffer (ROB), for example, two memory instructions cannot be simultaneously outstanding if they are 96 or more instructions away from each other, or if one depends on the other.

PDCM builds on this observation. It reconstructs the ROB during trace simulation to process a trace item only after the trace item enters the ROB and if the trace item does not have an unresolved dependency on preceding trace items in the ROB. In a nutshell, PDCM analyzes the ROB occupancy status to determine the progress of trace simulation and additionally when each trace item can issue a cache access.

3.1. PDCM setup

Our trace-driven simulation framework employs two distinct tools, a *trace generator* and a *trace simulator*. Fig. 2 illustrates the relationship between the trace generator, trace simulator, machine definitions, and trace files. The *generic machine definition* refers to the superscalar processor core configuration that we use in trace generation with a perfect L2 cache. The *target machine definition* is the system-level processor configuration such as L2 cache configuration and main memory latency, that completes the overall machine model we wish to study.

Following our previous work [18], we introduce the notion of *timing* during the trace generation phase and embed time-

Author's personal copy

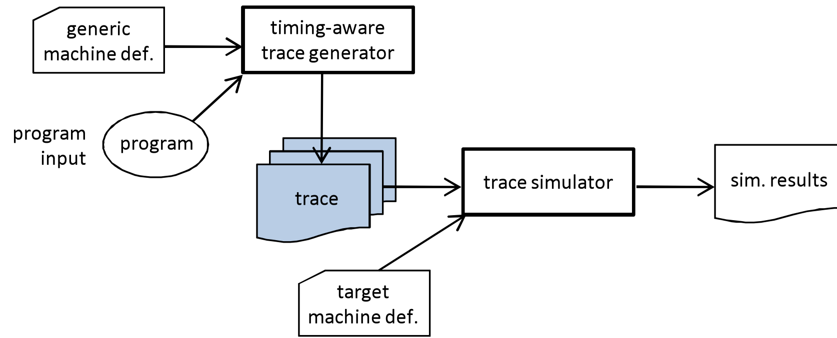


Fig. 2. Trace-driven simulation setup.

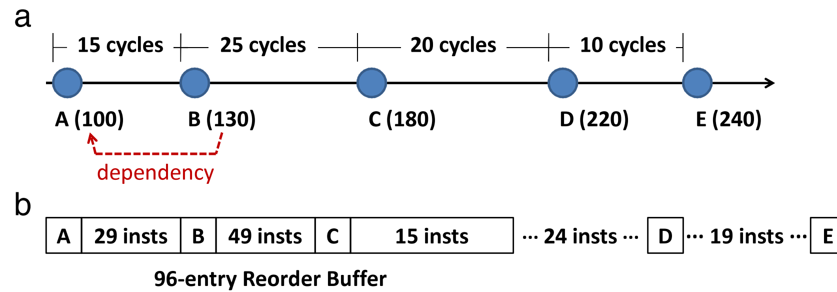


Fig. 3. (a) Five trace items (A, B, C, D, and E) recorded in the trace file. Trace item B depends on trace item A, while all other trace items are independent of each other. Inside parentheses is the instruction sequence number assigned to each trace item in program order. (b) The status of the ROB: Only the first three trace items are in the ROB.

related information in the trace. A trace item is generated from an L1 data cache miss, similar to [4]. Each trace item captures: the number of elapsed cycles since the last trace item and the information of the memory instruction that generated the trace item: cache access type (read or write), instruction sequence number, instruction sequence number of the parent trace item (if it depends on a previous trace item), cache address, and write-back address (if a write-back occurs on a data cache miss). To extract timing information, the trace generator must be able to model the microarchitecture of a processor using a user-provided machine definition. This work uses `sim-outorder` [1] to generate traces. The trace simulator implements our out-of-order trace simulation algorithm and the uncore component models.

In our PDCM framework, the processor core configuration used in the trace generator must be identical to the core configuration of the simulated machine. We assume 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. Identifying the data dependency between traces

Identifying all data dependencies between trace items is critical for accurate filtered trace simulation. In trace generation, to detect the dependencies between trace items, we construct and exploit data dependency chains during trace generation. In the dependency chain, the instruction sequence number of a parent trace item is propagated to the dependent trace items. We use the dependency chains already implemented in the modified `sim-outorder` simulator we use for trace generation. We note that a single trace item may depend on multiple trace items. However, we found that storing more than a single parent does not produce noticeably better results [18]. Accordingly, we choose to store at most one parent, the parent with the largest instruction sequence number, in each trace item.

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 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 caused the delayed hit, since there exists an implicit dependency between the two L1 data cache misses via the L1 delayed hit in between [5].

To expose all dependencies between trace items during the trace simulation phase, we generate trace items 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, we mark it with the instruction sequence number of the memory instruction that generated the miss. We assume a hit in L1 data cache is a delayed hit, if the difference between the instruction sequence number of the corresponding memory instruction and the instruction sequence number recorded in the cache block is smaller than a specified range.[3]

3.3. ROB occupancy analysis

ROB occupancy analysis examines the ROB occupancy status to determine the progress of the trace simulation. More specifically, we continue to run the trace simulation if the difference between the instruction sequence number of a trace item and the head of the ROB is smaller than the ROB size. A trace item can be processed if it is in the ROB and if it has no unresolved data dependency. Let us turn to the example in Fig. 3.

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

[3] The specified range should be at least the ROB size [5].

Author's personal copy

Table 1
Notations used in Section 3.4.

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

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.

Trace items generated for delayed hits 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 this case, trace item D cannot enter the ROB after trace item A commits, because the memory instruction 120 will become the head of the ROB.

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. Modeling a superscalar processor

Table 1 lists the notations used in this section.

3.4.1. Reconstructing the ROB

During trace simulation, we reconstruct the ROB 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 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 our experiments, we continued to fetch trace items 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 our ROB example in Fig. 3, 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.

3.4.2. Out-of-order trace simulation

We determine when to process a trace item (*traceProcessTime*) based on our 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, we analyze the ROB occupancy status and exploit the recorded cycle count 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 we estimate *traceProcessTime* of a trace item, we insert the trace item in a linked list that we will call *issue-list*. *issue-list* sorts the trace items in increasing order based on their *traceProcessTime*. In trace simulation, we process the trace item in the head of *issue-list* (*issueHead*), which has the smallest *traceProcessTime*. After *issueHead* is processed, we remove *issueHead* from *issue-list* and the next trace item in the list becomes the new *issueHead*.

Fig. 4 illustrates step by step how the trace items in the example of Fig. 3 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*, we set their *robReadyTime* 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, we cannot estimate B's *traceProcessTime* until we know A's *resolveTime*. 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 we set A's *resolveTime* to $N+$ memory access latency. After processing A, we remove A from *issue-list* and C becomes the new *issueHead*. Since we know A's *resolveTime* now, we can set B's *traceProcessTime* and insert B 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. We studied both approaches and observed that the lazy approach achieves higher accuracy on average than the eager approach. We show our results using both approaches in Section 5.1.

Continuing with our example, C is processed in cycle $N+40$, and C's *resolveTime* is set to $N+40+$ memory access latency. After processing C, we remove C, and B becomes the new *issueHead*. At cycle $N+$ memory access latency, we remove A 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.

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

3.4.3. PDCM with the baseline configuration

Fig. 5 shows the main loop in our 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 $(\cdot)$ notation to represent the association between a trace item and the recorded information. For example, *robHead.ISN* means *ISN* of *robHead*.

- *ReconstructROB()*: Fig. 6 describes how we reconstruct the ROB in trace simulation. When the simulated clock cycle time (*current_*

Author's personal copy

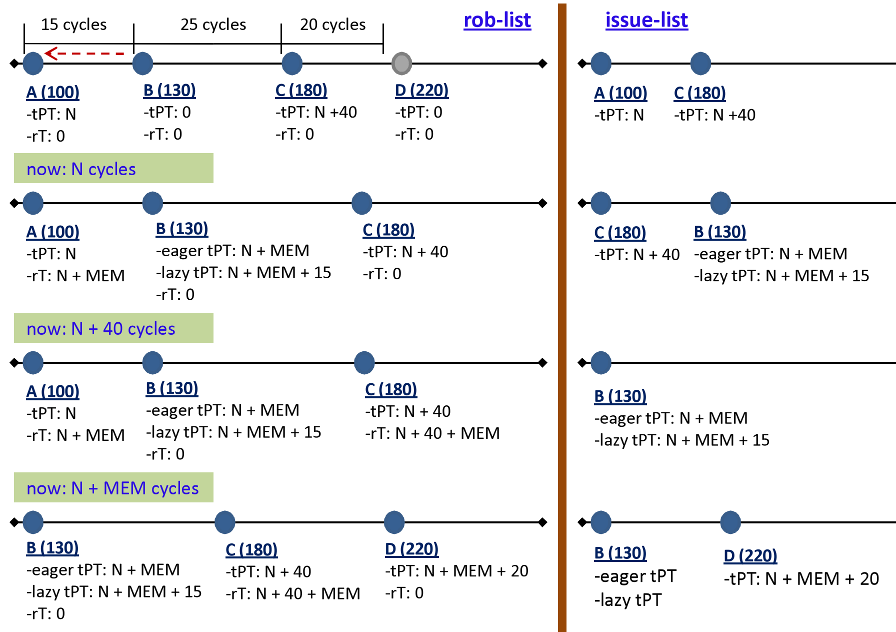


Fig. 4. Using *rob-list* and *issue-list* to reconstruct the ROB and issue L2 cache accesses out of order during trace simulation. The example builds on the ROB example in Fig. 3. The red dotted arrow shows that trace item B depends on A. The cycle counts between trace items on the first row assume perfect 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
```

Fig. 5. 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.

time) reaches the time to commit a trace item (*next_commit_time*), we attempt to remove the trace item from *rob-list* (lines 2–9). If *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, we make a write access to the L2 cache before it is removed (lines 3–5). We commit the trace items 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, we fetch new trace items from the trace file (lines 16–26). After a trace item is inserted in *rob-list*, we insert the trace item in *issue-list* if we can estimate the trace item's *traceProcessTime*.

• *ProcessTraceItem()*: The L2 cache is accessed by *issueHead*, as described in Fig. 7. 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. We set *traceProcessTime* of the identified dependent trace items and insert them in *issue-list* (lines 6–12). After processing *issueHead*, the next trace item in *issue-list* becomes the new *issueHead* (line 16), and we determine when to process the new *issueHead* (line 17).

If *issueHead* is a write trace item, we do not access the L2 cache, but we set *issueHead*'s *resolveTime* to *current_time*. The write trace item accesses the L2 cache when it commits from *rob-list*.

3.4.4. Modeling various processor artifacts in PDCM

The algorithm described above can be easily extended to model important processor artifacts, such as MSHRs, data prefetching, branch mis-predictions, and instruction caching. To add new processor artifacts in the analytical models [5,7,14,22], 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 mis-prediction 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 MSHRs*: In this paper, we focus on MSHRs for L2 cache misses; 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, we change *issueHead*'s *traceProcessTime* to the time when an MSHR becomes available and reorder *issue-list*.

• *Modeling a data prefetcher*: In our work, we model a tagged prefetcher. The tagged prefetcher fetches the next sequential cache block when a miss occurs, or when a hit occurs in a prefetched block [25]. 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. However, some trace items are generated from L1 data cache hits because we generated extra trace items for the delayed hits during trace generation. Hence, if the L2 cache accesses from the delayed hit trace items hit in the prefetched blocks, the tagged prefetcher will make more prefetching requests than it should. To correctly handle data prefetching, we distinguish the data cache miss trace items from the delayed hit trace items. In trace simulation, we monitor all L2 cache misses and the L2 hits only from the data cache miss trace items.

Author's personal copy


```
1:  robNode = NULL;
2:  while (robHead is not NULL) and (robHead.resolveTime > 0) and (current_time >
    robHead.resolveTime) do
3:     if (robHead is a write trace item) then
4:        Issue a write access to L2 cache;
5:     end if
6:     robNode = robHead.next; /*next trace item in rob-list*/
7:     Commit robHead; /*remove robHead from rob-list*/
8:     robHead = robNode;
9:  end while
10: while (robNode is not NULL) and
    (robNode.ISN − robHead.ISN < ROB size) do
11:    if (robNode.pending == TRUE) then
12:       robNode.pending = FALSE; /* insert pending trace items in rob-list */
13:    end if
14:    robNode = robNode.next;
15: end while
16: while (1) do
17:    newTrace = new trace item from the trace file;
18:    if (newTrace.ISN − robHead.ISN < ROB size) then
19:       insert newTrace in rob-list;
20:    else if (newTrace.ISN − robHead.ISN ≥ ROB size) and
       (newTrace.ISN − robHead. ISN < 2× ROB size) then
21:       newTrace.pending = TRUE;
22:       insert newTrace in rob-list;
23:    else
24:       break;
25:    end if
26: end while
```

Fig. 6. High-level pseudo-code for reconstructing the ROB.

```
1:  rob = issueHead's corresponding trace item in rob-list;
2:  if (issueHead is a read trace item) then
3:     make read access to L2 cache;
4:     rob.resolveTime = current_time + L2 access latency;
5:     node = robHead;
6:     while (node is not NULL) do
7:        if (node's parent.ISN == issueHead.ISN) then
8:           node.traceProcessTime =
                MAX(node.robReadyTime,
                    current_time + L2 access latency + elapsed cycles between issue-
           Head and node);
9:           Insert node in issue-list;
10:       end if
11:       node = node.next; /*next trace item in rob-list*/
12:    end while
13: else
14:    /* IssueHead is a write trace item*/
       rob.resolveTime = current_time;
15: end if
16: issueHead = issueHead.next; /*next trace item in issue-list*/
17: next_event_time = issueHead.traceProcessTime;
```

Fig. 7. High-level pseudo-code for processing a trace item.

• *Modeling branch prediction*: Branch mis-predictions during trace generation incur "speculative" trace items on mis-predicted control paths. In trace generation, we distinguish the speculative trace items from the non-speculative 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, we remove the speculative trace item from *rob-list* after it is processed.

We observed two important aspects that can affect the accuracy of branch handling. 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 Fig. 8 for illustration.

In Fig. 8(a), assume instruction 100 is the head of the ROB and a branch mis-prediction 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 in our analysis, ISN is incremented only when it is assigned to the non-speculative instructions.

Our 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. Fig. 8(b) shows an example from *mcf* benchmark in SPEC2K benchmark suite [26], where a branch instruction depends on memory instructions. To address this issue, we generate an extra

Author's personal copy

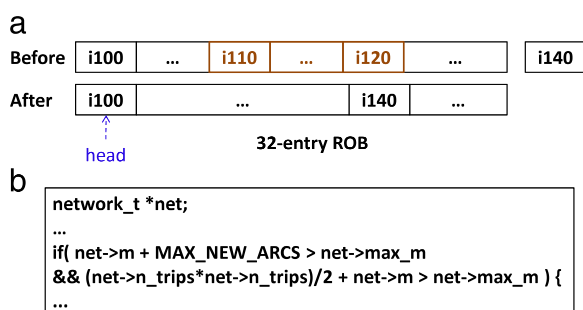


Fig. 8. (a) The ROB occupancy status before and after the branch mis-prediction is resolved. Assume instructions 110 through 120 are fetched from an incorrectly predicted control path. (b) An example from *mcf* where a branch depends on memory instructions.

trace item for a branch instruction 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 5.1, we show that our approach accurately models the effect of branch mis-predictions.

• *Modeling instruction caching*: Finally, to model the effect of instruction caching, we employ a realistic instruction cache in trace generation. If L2 cache is accessed by an L1 instruction cache miss, we allow an L2 cache miss to occur in trace generation. The timing information is recorded in the trace and exploited in trace simulation.

3.5. Discussions

Before we evaluate our approach, we summarize and discuss the limitations of PDCM in the following.

• PDCM requires an execution-driven simulator that models the target processor to generate timing-aware filtered traces.

• PDCM requires an initial simulation to generate traces. Evidently, our approach is not practical when running a simulation one time only. However, the trace generation time is amortized as we run many simulations reusing the generated trace. We note that it is typical to run numerous simulations when studying the trade-offs of different uncore configurations.

• PDCM focuses only on assessing the impact of uncore events, such as L1 cache misses, on program execution time and assumes that a superscalar processor core's parameters are fixed during a series of uncore experiments. Hence, the traces have to be regenerated if the core parameter is changed.

4. Experimental setup

4.1. Machine configurations and benchmarks

Table 2 lists our "baseline" and "realistic" superscalar processor configurations. The configurations are intended to resemble the Intel Core 2 Duo processor [11]. However, the baseline configuration does not incorporate any processor artifacts. We will first demonstrate that our model is accurate with the baseline configuration. We then further evaluate our model by individually adding a key superscalar processor artifact to the baseline configuration in consideration. Finally, we will use the realistic superscalar processor configuration incorporating all the artifacts. We use the entire SPEC2K benchmarks to evaluate our model. The inputs for benchmarks are listed in Table 3.

Programs were compiled using the Compaq Alpha C compiler (V5.9) with the −O3 optimization flag. For each simulation, we skip the initialization phase of the target program [24], warm up caches in the next 100M instructions, and simulate the next 1B instructions.

Table 2
Baseline and realistic superscalar processor configurations.

	Baseline	Realistic
Dispatch/issue/commit width	4	
Reorder buffer	96 entries	
Load/Store queue	96 entries	
Integer ALUs	4	
Floating point ALUs	2	
L1 d-cache	2 cycles, 32 kB, 8-way 64B line size, LRU	
L1 i-cache	Perfect	Same as L1 d-cache
L2 cache (unified)	12 cycles, 2 MB, 8-way 64B line size, LRU	
Main memory latency	200 cycles	
Branch predictor	Perfect	Combined · bimodal and gshare · 4 K meta-table size
L2 MSHRs	Unlimited	8
L2 data prefetcher	–	Tagged prefetcher

Table 3
Inputs for the SPEC2k benchmarks used in experiments.

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

4.2. Metrics

To demonstrate the efficacy of PDCM, we employ a PDCM-based trace-driven simulator (PDCM) and compare it with a detailed execution-driven simulator (`sim-outorder`). Our main metrics are *CPI error* and *relative CPI change*. CPI error is defined as $(CPI_{pdcm} - CPI_{soo})/CPI_{soo}$, where CPI_{pdcm} and CPI_{soo} are the CPI obtained with PDCM and `sim-outorder`, respectively. The CPI error of PDCM shows the difference in cycle count compared with `sim-outorder`. The absolute value of the CPI error shows the magnitude of the difference. The average CPI error is obtained by taking the arithmetic mean of the absolute CPI errors, as similarly defined in [5]. 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 [21]. We use relative CPI change to measure the performance change of the revised configuration relative to the performance of the baseline configuration.

To compare the performance change amount shown by `sim-outorder` and PDCM, we use *relative CPI error*. We define relative CPI error as $|rel_cpi_chg_{soo} - rel_cpi_chg_{pdcm}|$, where $rel_cpi_chg_{soo}$ and $rel_cpi_chg_{pdcm}$ are the relative CPI change reported by `sim-outorder` and PDCM, respectively.

Lastly, we evaluate PDCM's capability to reproduce the behavior of a superscalar processor by tracking the temporal changes in memory access patterns. We divide a program execution into intervals and generate a histogram to collect the frequency of the distance (in cycles) between two consecutive memory accesses in each interval. We compare the frequency of the collected distances in PDCM and `sim-outorder` to examine how closely PDCM reproduces the off-core memory access patterns of `sim-outorder`.

Author's personal copy

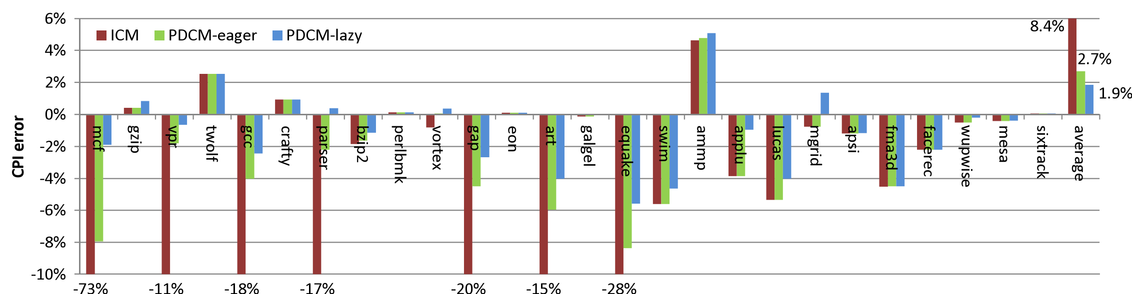


Fig. 9. The CPI errors of the entire SPEC2K benchmarks using the baseline configuration. "ICM" is the independent cache miss model, and "PDCM-eager" and "PDCM-lazy" stand for eager and lazy estimation of the dependent trace item's processing time.

5. Evaluation results

5.1. Accuracy of PDCM

• *PDCM with the baseline configuration*: Fig. 9 presents the CPI error of the SPEC2k benchmarks with the baseline configuration. The CPI error of the independent cache miss model (ICM) [18]—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 3.4.2) 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, we will use the lazy approach 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 we do not take into account the overlap between an L2 cache miss 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, we simply used the entire cycle count to estimate the processing time of the pending trace item in trace simulation. We 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 5.3). What is more important at this point is the magnitude, which is fairly small.

To show that PDCM is robust to the variation in processor's inherent parameters, we used different ROB size, L1 data cache size, and issue-width to study the sensitivity of our model. Our results show that the CPI errors were less than 3% when we used different processor core configuration in trace generation. Table 4 summarizes our study results.

Although we used a fixed main memory access latency in our experiments, the accuracy of PDCM does not depend on the off-chip access latency. We ran experiments with a DRAM model in sim-outorder and PDCM that has 16 banks (8 banks × 2 ranks) with 16 kB row size and assumed an open-page policy. We assumed the main memory access latency is 80 cycles when a hit occurs in the row-buffer, and 180 cycles when a conflict occurs in the row-buffer. The average CPI error of PDCM was 4.1% over the entire SPEC2k benchmarks.

Table 4
The accuracy of PDCM with different processor core configurations.

Different issue-width				
Width		2		8
Avg. CPI error (%)		2.2		2.1
Different L1 data cache sizes				
Size	8 kB		16 kB	64 kB
Avg. CPI error (%)	2.5		2.1	1.8
Different ROB sizes				
Size	32	64	128	256
Avg. CPI error (%)	1.5	1.9	2.3	2.8

Finally, we evaluated the accuracy of PDCM using a combination of different processor core configurations and the DRAM model described above. We assumed the processor core has 8 kB L1 data cache, 2 issue-width, and 256 entries in ROB. We configured sim-outorder accordingly, and collected the filtered traces using the same core configuration for PDCM. The average CPI error of PDCM was 3.3% over the entire SPEC2k benchmarks, which shows that the accuracy of PDCM is not affected by different processor core configurations and a realistic memory access latency.

In what follows, we evaluate PDCM 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 limited MSHRs*: Fig. 10 compares the relative CPI change obtained with sim-outorder and PDCM, when limited number of MSHRs is applied to the baseline configuration. Since the number of outstanding cache misses is limited by the number of MSHRs, the CPI increases with fewer MSHRs.

The results show that PDCM can closely follow the relative CPI change of sim-outorder. The relative CPI error of the benchmarks in the figure was 2% on average. *fma3d* has a very high L2 cache miss rate and it is particularly sensitive to the number of MSHRs. We observed that *fma3d*'s 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.

• *Effect of data prefetching*: Fig. 11 compares the relative CPI change reported by sim-outorder and PDCM, when a tagged prefetcher is added in the baseline configuration. The results show that PDCM can accurately model the effect of data prefetching. The two largest beneficiaries were *swim* and *mgrid* as shown by both sim-outorder and PDCM. The relative CPI error 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.

Author's personal copy

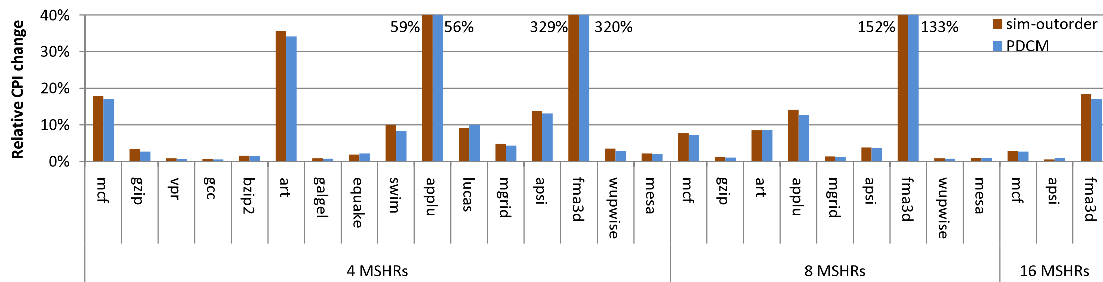


Fig. 10. The relative CPI change when limited number of MSHRs (4, 8, and 16) is applied to the baseline configuration. Only the benchmarks that show at least 1% relative CPI change are presented in the figure.

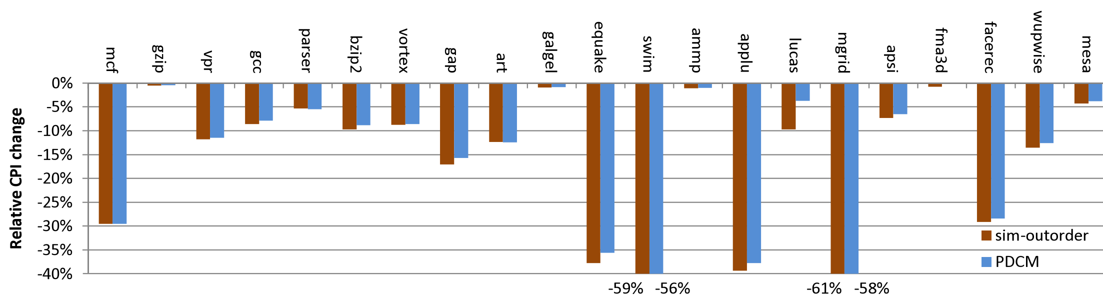


Fig. 11. The relative CPI change when a tagged prefetcher is incorporated in the baseline configuration.

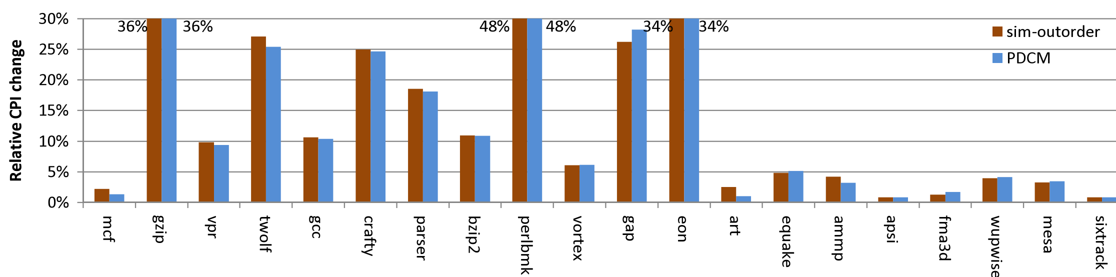


Fig. 12. The relative CPI change after incorporating a realistic branch predictor in the baseline configuration.

• *Effect of branch prediction*: The trace files used in the previous sections are generated with a perfect branch predictor. In this section, we 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. We configured `sim-outorder` with the identical branch predictor.

Fig. 12 compares the relative CPI change shown by `sim-outorder` and PDCM. The largest branch mis-prediction penalties was shown in *perl* from both `sim-outorder` and PDCM. The relative CPI error of the benchmarks presented in the figure was 1% on average. The CPI error after incorporating a realistic branch predictor was 1.8% 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 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, we cannot correctly model the branch mis-prediction penalties using the cycle counts in the trace items. To investigate this aspect, we quantified the effect of the memory access latency on branch prediction accuracy using eight representative benchmarks in the SPEC2k benchmark suite [13], as shown in Table 5.

We collect the branch prediction accuracy separately from two simulation runs. The 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), we collect the number of correct branch predictions from A with a fixed latency ($n_correct_{A-fixedlat}$) and a random latency ($n_correct_{A-randomlat}$), and the number of total branch predictions made from A (n_total_A). The *stability* of a given branch instruction A is defined by:

$$stability = \frac{|n_correct_{A-fixedlat} - n_correct_{A-randomlat}|}{n_total_A}.$$

For instance, the last column in Table 5 shows that 88% of *mesa*'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.

Table 5
The percent of stable branch instructions in the benchmarks.

Stability	% of stable branch instructions in the program			
	≤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

Author's personal copy

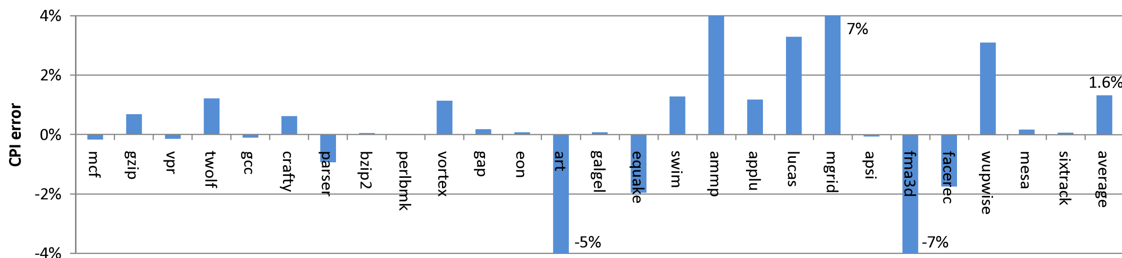


Fig. 13. The CPI errors of the SPEC2k benchmarks using the realistic processor configuration.

• *PDCM with the realistic configuration*: We finally present the simulation results for the realistic superscalar processor configuration with PDCM. The trace files used in this section are generated with branch mis-predictions and instruction cache misses. We also modified `sim-outorder` to model the realistic superscalar processor. The CPI errors of the SPEC2k benchmarks are reported in Fig. 13. 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.

5.2. Reproducing temporal uncore access behavior

We have previously used CPI error and relative CPI change computed over the entire execution span as the main metrics to evaluate how closely PDCM approximates a realistic superscalar processor's performance. In this and the next subsection, we will focus on two aspects of PDCM that are relevant at the system level: (1) Does PDCM change 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, we build histograms of the distance (in cycles) between two consecutive memory accesses (from L2 cache misses, write backs, or L2 data prefetching) 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. We assume that PDCM preserves the temporal memory access patterns of `sim-outorder` if the frequency of distances between two consecutive memory 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. We use the metric $Similarity = \frac{\sum_{i=0}^{n} \text{MIN}(bin_soo_i, bin_pdcm_i)}{\sum_{i=0}^{n} bin_soo_i}$ in order 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 memory access pattern of `sim-outorder`. If the similarity is 1, it suggests that the observations made by the two simulators are identical.

Fig. 14 depicts the representative interval of *mesa* and *parser*. *mesa* shows that `sim-outorder` and PDCM agree well on the memory access behavior, while *parser* shows that `sim-outorder` and PDCM disagree somewhat on the frequency of the distances between isolated memory accesses. Table 6 presents the computed average *Similarity* over all intervals for all SPEC2k benchmarks. We 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 we do not take the first two memory accesses with PDCM 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.

Table 6
The similarity in memory access patterns between `sim-outorder` and PDCM (shown in percentage).

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%)

5.3. Predicting the performance with different uncore parameters

We 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. We make our realistic superscalar processor configuration as the reference point and simulate five configurations that differ in one of their L2 cache or main memory parameters as shown in Table 7. In Configuration 1 and 2, the L2 cache is 1 MB and 4 MB instead of 2 MB ("smaller L2 cache" and "larger L2 cache"). In Configuration 3 and 4, the memory latency is 150 cycles and 300 cycles instead of 200 cycles ("faster memory" and "slower memory"). In Configuration 5, the L2 hit latency is 20 cycles instead of 12 cycles ("slower L2 cache"). Note that PDCM used the same traces produced to study the realistic superscalar processor configuration in Table 2 for all five different configurations. On the other hand, we ran `sim-outorder` with each new machine configuration examined.

The results in Table 7 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 7 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 error when L2 cache size was reduced (Conf. 1). *gcc*'s CPI increased 78% in `sim-outorder` when L2 cache size is reduced from 2 MB to 1 MB, whereas PDCM increased CPI by only 16%. PDCM shows smaller CPI increase because it has smaller L2 cache misses with 1 MB 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 the end of Section 3.4.4. Our simple approach does not accurately capture the case when L1

Author's personal copy

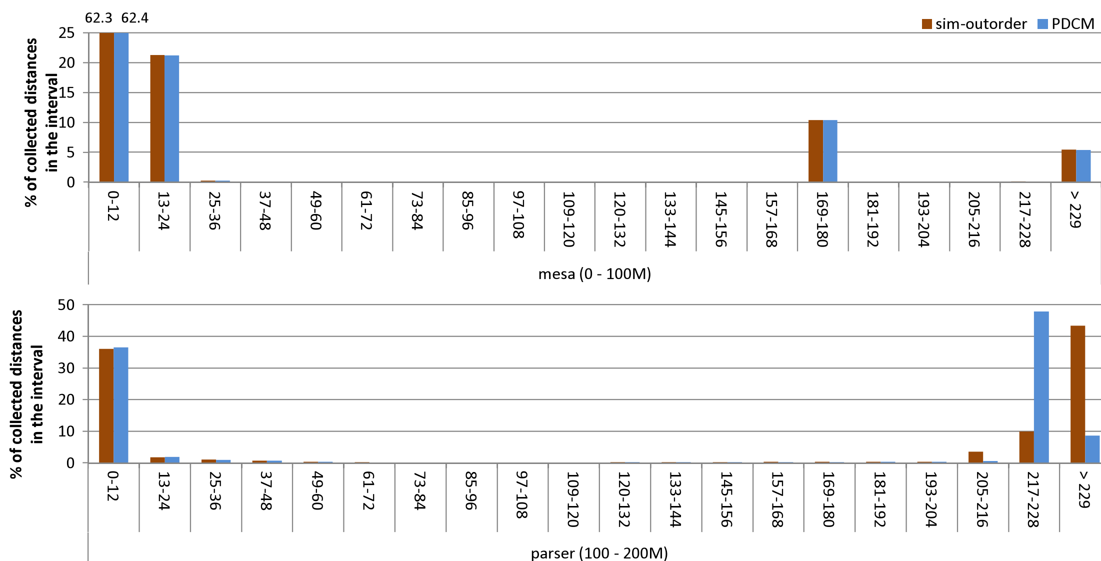


Fig. 14. The histogram of collected distances in an interval is depicted to compare the distance (in cycles) between two consecutive L2 cache misses in `sim-outorder` and PDCM. The *x*-axis represents the bins used to generate the histogram and the *y*-axis represents the percent of collected distances in the interval of 100M instructions. The bin size is 12 cycles. We only show the representative intervals of the benchmarks that show the highest (*mesa*) and lowest (*parser*) similarity for clear presentation.

Table 7
Relative CPI error between `sim-outorder` and PDCM. The five configurations are identical to the baseline configuration (Table 2) except a single parameter.

	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

instruction cache misses increase the number of L2 cache misses with smaller L2 cache. In general, we were able to obtain fairly accurate projections of the relative performance from our model. The average relative CPI error of the five configurations ranged from 0.3% (larger L2) to 3.4% (smaller L2).

In summary, the results presented in Fig. 14, Tables 6 and 7 suggest that PDCM is amenable for use in a multicore simulation environment [20,19]. To simulate multiple processor cores that run independent threads simultaneously (i.e., multiprogrammed workload), one can prepare traces from a detailed uniprocessor simulator (like `sim-outorder`) and run them together. Our techniques can be applied to multithreaded shared memory applications if individual threads can be traced [19]. One can reliably study the overall system behavior thanks to the capability of our 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.

5.4. Simulation speed and storage requirements

Lastly, we report the simulation speed and storage requirements of PDCM and a full trace-driven simulation strategy, a widely practiced simulation method [2]. 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 we model the full detail of the target superscalar processor operation per every supported instruction, our implementation of the full trace simulator is essentially identical to `sim-outorder`. The observed simulation speedups[4] of PDCM range from 3.8× (*gcc*) up to 582.58× (*eon*) and their average (geometric mean) was 62.5×. 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×.

In our 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 12.3 GB. A single trace item with PDCM is 20B, and the average trace file size is 1.5 GB, ranging from 20 MB (*eon*) to 9.6 GB (*gcc*). 22 out of 26 benchmarks were less than 1.9 GB, and 14 out of 26 were less than 1 GB. Certain benchmarks require many trace items because of delayed hits, especially *gcc*. Since not all delayed hits may be needed for accuracy, the trace file size can be reduced significantly if we filter unnecessary delayed hits during

[4] 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 SPEC2k benchmarks.

Author's personal copy

trace generation as a possible optimization. Trace file sizes were slightly larger with the realistic configuration with an average of 1.5 GB, and range from 25 MB (*eon*) to 9.7 GB (*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 we discussed in this section. Considering the fast simulation speed, small error, and much smaller storage overheads, we believe PDCM is a more attractive simulation method than the full trace-driven simulation, especially in the early design stages.

6. Related work

Trace-driven simulation has been an indispensable technique for analyzing computer performance. Our work is a positive response to the question "Can trace-driven simulators accurately predict superscalar performance?" [3]. In their work, Black et al. showed that sampling techniques present a problem to the accuracy of trace-driven simulation for superscalar processors. This paper advocated using timing-embedded filtered trace to model superscalar processor performance.

In previous and current practice, much trace-driven simulation work has focused on either tracing memory Refs. [27] or using a full trace of executed instructions for relatively fast simulation with complete fidelity [2]. In our previous works [18,17], we demonstrated that filtered trace-driven simulation can accurately approximate superscalar processor performance. In [18], we introduced three different trace-driven simulation models using filtered traces and discussed the strength and weakness of each 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 a more recent work [17], we 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. In-N-Out uses a processor model that is more abstract than this work—still more concrete than other proposals that we will discuss below. Hence, In-N-Out is less accurate than PDCM, but can attain potentially higher simulation speeds.

Michaud et al. [22] built an analytical model to study the relations between instruction fetching, branch prediction accuracy, and ILP. Their model is based on the observation that the number of instructions that can issue per cycle grows as the square root of the instruction window size. However, they assumed an idealized superscalar processor, whereas our target processor is a realistic superscalar processor. Karkhanis and Smith [14] built an analytical model that divides program execution into intervals separated by "miss-events" like branch mis-prediction, instruction cache miss, and data cache miss. The overall performance of a program is computed with the baseline CPI (measured with no miss-events) and the CPI due to miss-events. They introduced the idea of exploiting the ROB size and the distance between memory instructions to model the performance penalty of L2 cache misses. Our work was in part inspired by their analytical performance model. However, their analytical model did not model the effect of limited MSHRs and data prefetching. In contrast, PDCM can accurately model the effect of limited MSHRs and data prefetching.

Chen and Aamodt [5] improved on the model of [14] 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 [14]. However, they assumed that both branch predictor and instruction cache are ideal, and their simulator has to support prefetching when generating the instruction trace to model the prefetching effect. Eyerman et al. [7] proposed their "mechanistic model", which is the new generation of the first order model [14]. They describe the details of the interval analysis proposed by Karkhanis and Smith and the revised model. Similar to Karkhanis and Smith, they modeled branch prediction, data cache misses, and instruction cache misses, but did not consider the limited MSHRs or data prefetching. Genbrugge et al. [10] extended the mechanistic model to raise the level of abstraction for fast multicore simulation.

The analytical models [22,14,5,7] and PDCM all use trace files as an input to the model for different purposes. The analytical models analyze a full instruction trace before using their model to get statistics for their mathematical equations. In our work, we use a filtered trace for trace reduction and we do not need to analyze the trace file before the trace simulation. PDCM considers the timing and dependency information captured during trace generation and focuses on fast dynamic simulation rather than hybrid analytical modeling. Analytical models provide well-constructed mathematical equations, while PDCM provides a novel trace simulation framework for quick and accurate performance modeling. In our work, we studied the temporal uncore access behavior of a superscalar processor with our model. Such study was not conducted by the analytical models.

While not directly comparable to our approach, there are other interesting works done to reduce the simulation time overhead. 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. [8] developed a framework that generates synthetic clones for the target benchmarks using the characterized information of the benchmarks. On the other hand, MinneSPEC tries to reflect the behavior of SPEC2k benchmarks while using a smaller, but representative input set [15]. Genbrugge and Eeckhout [9] 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.

Sampling techniques have been developed to reduce the scope of (detailed) simulation during execution-driven simulation. Sherwood et al. [24] 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 [24], 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 $\sim$100 (100 M per phase $\times$ 10 phases/100 B instructions). The average IPC error of SimPoint based simulation, compared with sim-outorder, was reported to be $\sim$3% over the SPEC2k benchmarks. Another sampling method, SMARTS, was proposed by Wunderlich et al. [28]. Unlike SimPoint that reduces the scope of detailed simulation to specific phases, SMARTS systematically samples program execution intervals with relatively fine granularity without paying attention to program behavior changes. The number of samples and the length of each sample depend on the desired target confidence level. Compared with sim-outorder, SMARTS was shown to achieve 60$\times$ simulation speedup and less than 1% error (on average). Compared with SimPoint and SMARTS, PDCM resorts to cache filtering, a fundamentally different sampling strategy with no bearing on simulation intervals. PDCM offers 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, PDCM could work together with either SimPoint or SMARTS (in the context of trace-driven simulation).

Author's personal copy

Chou et al. [6] 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.

Finally, there are trace-driven multicore simulators, TPTS [19] and Zauber [20], that also use timing-embedded filtered traces. We expect that our model can be easily integrated in such trace-driven multicore simulators.

7. Conclusions

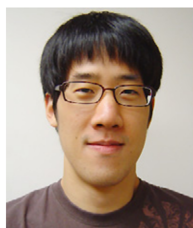
The pairwise dependent cache miss model (PDCM) enables a trace-driven superscalar processor simulation method using reduced traces. We use PDCM to study how the out-of-order instruction execution capability of a superscalar processor, as a function of the ROB size, affects the trace-driven simulation methodology. We focus on assessing the impact of uncore events on program execution times and assume that a superscalar processor's parameters are fixed during a series of experiments. We make the following contributions:

- We develop techniques to incorporate the effect of key processor architecture artifacts on the program execution time and model a realistic superscalar processor incorporating those artifacts at length using reduced trace.
- Compared with a full trace-driven simulation method that incurs large simulation time and storage overheads, our work uses storage efficient filtered traces that carry enough information to compute latencies between trace items at simulation time. Our approach obtains competitive or better accuracy than previously published simulation methods that use memory traces, both filtered and unfiltered.
- Compared with a detailed execution-driven simulation method, PDCM achieves a simulation speedup of 62.5× on average (geometric mean) while giving sufficiently small errors across benchmarks (less than 3% on average).
- PDCM is an attractive simulation method especially in early processor design stages. It 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.

As current and future processor research is centered on multicore architectures, the importance of studying system-wide resources will continue to grow. Our study forms a basis for tools that enable fast and accurate multicore simulations focusing on system-wide resources, such as shared L2 cache, on-chip network, and memory controller.

References

[1] T. Austin, E. Larson, D. Ernst, Simplescalar: an infrastructure for computer system modeling, IEEE Computer 35 (2) (2002) 59–67.
[2] L. Barnes, Performance modeling and analysis for AMD's high performance microprocessors, in: Keynote at Int'l Symp. Performance Analysis of Systems and Software, ISPASS, 2007.
[3] B. Black, A.S. Huang, M.H. Lipasti, J.P. Shen, Can trace-driven simulators accurately predict superscalar performance? in: Proc. Int'l Conf. Computer Design, ICCD, 1996, pp. 478–485.
[4] J. Chame, M. Dubois, Cache inclusion and processor sampling in multiprocessor simulations, in: Proc. ACM SIGMETRICS Conf., 1993, pp. 36–47.
[5] X. Chen, T. Aamodt, Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs, in: Proc. Int'l Symp. Microarchitecture, MICRO, 2008, pp. 455–465.
[6] Y. Chou, B. Fahs, S. Abraham, Microarchitecture optimizations for exploiting memory-level parallelism, in: Proc. Int'l Symp. Computer Architecture, ISCA, 2004, pp. 76–87.
[7] S. Eyerman, L. Eeckhout, T. Karkhanis, J.E. Smith, A mechanistic performance model for superscalar out-of-order processors, ACM Transactions on Computer Systems 27 (2) (2009) 1–37.
[8] K. Ganesan, J. Jo, 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, 2010, pp. 33–44.
[9] D. Genbrugge, L. Eeckhout, Memory data flow modeling in statistical simulation for the efficient exploration of microprocessor design spaces, IEEE Transactions on Computers 57 (10) (2008) 41–54.
[10] D. Genbrugge, S. Eyerman, L. Eeckhout, Interval simulation: raising the level of abstraction in architectural simulation, in: Proc. Int'l High-Performance Computer Architecture, HPCA, 2010, pp. 307–318.
[11] Intel Corp., Intel 64 and IA-32 Architectures Software Developer's Manual, Vol 3A: System Programming Guide, Part1, 2010.
[12] M. Johnson, Superscalar Microprocessor Design, Prentice Hall, 1991.
[13] A. Joshi, A. Phansalkar, L. Eeckhout, L.K. John, Measuring benchmark similarity using inherent program characteristics, IEEE Transactions on Computers 55 (6) (2006) 769–782.
[14] T. Karkhanis, J.E. Smith, The first-order superscalar processor model, in: Proc. Int'l Symp. Computer Architecture, ISCA, 2004, pp. 338–349.
[15] A. KleinOsowski, D.J. Lilja, MinneSPEC: a new SPEC benchmark workload for simulation-based computer architecture research, Computer Architecture Letters (CAL) 1 (2002).
[16] D. Kroft, Lockup-free instruction fetch/prefetch cache organization, in: Proc. Int'l Symp. Computer Architecture, ISCA, 1981.
[17] K. Lee, 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, 2011, pp. 126–135.
[18] K. Lee, S. Evans, S. Cho, Accurately approximating superscalar processor performance from traces, in: Int'l Symp. Performance Analysis of Systems and Software, ISPASS, 2009, pp. 238–248.
[19] H. Lee, L. Jin, K. Lee, S. Demetriades, M. Moeng, S. Cho, Two-phase trace-driven simulation (TPTS): a fast multicore processor architecture simulation approach, Software: Practice and Experience (SPE) 40 (3) (2010) 239–258.
[20] Y. Li, B. Lee, D. Brooks, Z. Hu, K. Skadron, CMP design space exploration subject to physical constraints, in: Proc. Int'l Symp. High-Performance Computer Architecture, HPCA, 2006, pp. 62–72.
[21] D.J. Lilja, Measuring Computer Performance: A Practitioner's Guide, Cambridge University Press, 2000.
[22] P. Michaud, A. Seznec, S. Jourdan, An exploration of instruction fetch requirement in out-of-order superscalar processors, International Journal of Parallel Programming 29 (1) (2001) 35–58.
[23] J.P. Shen, M.H. Lipasti, Modern Processor Design: Fundamentals of Superscalar Processors, McGraw-Hill, 2004.
[24] T. Sherwood, E. Perelman, G. Hamerly, B. Calder, Automatically characterizing large scale program behavior, in: Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems, ASPLOS, 2002, pp. 45–57.
[25] A.J. Smith, Cache memories, ACM Computing Surveys 14 (3) (1982) 473–530.
[26] Standard Performance Evaluation Corporation. http://www.spec.org.
[27] R.A. Uhlig, T.N. Mudge, Trace-driven memory simulation: a survey, ACM Computing Surveys 29 (2) (1997) 128–170.
[28] R.E. Wunderlich, T.F. Wenisch, B. Falsafi, J.C. Hoe, SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling, in: Proc. Int'l. Symp. Computer Architecture, ISCA, 2003, pp. 84–95.
[29] K.C. Yeager, The MIPS R10000 superscalar microprocessor, IEEE Micro 16 (2) (1996) 28–40.
[30] J.J. Yi, L. Eeckhout, D.J. Lilja, B. Calder, L.K. John, J.E. Smith, The future of simulation: a field of dreams, IEEE Computer 39 (11) (2006) 22–29.



Kiyeon Lee received his B.S. degree in Computer Science and Technology from Tsinghua University in 2006. He has been pursuing a Ph.D. in Computer Science from the University of Pittsburgh since 2006. His research interests are in the area of computer architecture, with particular focus on simulation methodologies.

Sangyeun Cho received his B.S. degree in Computer Engineering from Seoul National University in 1994 and a Ph.D. in Computer Science from the University of Minnesota in 2002. In 1999, he joined the System LSI Division of Samsung Electronics Co., Giheung, Korea, and contributed to the development of Samsung's flagship embedded processor core family named CalmRISC(TM). He was a lead architect of CalmRISC-32, a 32-bit microprocessor core, and designed its memory hierarchy including caches, DMA, and stream buffers. Since 2004, he has been with the Computer Science Department at the University of Pittsburgh, where he is an Associate Professor. His research interests are in the area of computer architecture and embedded systems, with particular focus on performance, power, and reliability aspects of memory and storage hierarchy design for next-generation multicore platforms.