Per-Thread Cycle Accounting in Multicore Processors

KRISTOF DU BOIS, STIJN EYERMAN, and LIEVEN EECKHOUT, Ghent University

While multicore processors improve overall chip throughput and hardware utilization, resource sharing among the cores leads to unpredictable performance for the individual threads running on a multicore processor. Unpredictable per-thread performance becomes a problem when considered in the context of multicore scheduling: system software assumes that all threads make equal progress, however, this is not what the hardware provides. This may lead to problems at the system level such as missed deadlines, reduced quality-of-service, non-satisfied service-level agreements, unbalanced parallel performance, priority inversion, unpredictable interactive performance, etc.

This article proposes a hardware-efficient per-thread cycle accounting architecture for multicore processors. The counter architecture tracks per-thread progress in a multicore processor, detects how inter-thread interference affects per-thread performance, and predicts the execution time for each thread if run in isolation. The counter architecture captures the effects of additional conflict misses due to cache sharing as well as increased latency for other memory accesses due to resource and bandwidth contention in the memory subsystem. The proposed method accounts for 74.3% of the interference cycles, and estimates per-thread progress within 14.2% on average across a large set of multi-program workloads. Hardware cost is limited to 7.44KB for an 8-core processor, a reduction by almost 10× compared to prior work while being 63.8% more accurate. Making system software progress aware improves fairness by 22.5% on average over progress-agnostic scheduling.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General—*Modeling of computer architecture*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling techniques*

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Multicore processors, hardware/software interface, scheduling, resource sharing, interference, performance analysis

ACM Reference Format:
Du Bois, K., Eyerman, S., and Eeckhout, L. 2013. Per-Thread cycle accounting in multicore processors. ACM Trans. Architec. Code Optim. 9, 4, Article 29 (January 2013), 22 pages.
DOI = 10.1145/2400682.2400688 http://doi.acm.org/10.1145/2400682.2400688

1. INTRODUCTION

Multicore processors, or chip-multiprocessors (CMPs), seek at increasing chip throughput within a given power budget by integrating multiple cores on a single chip. Processor manufacturers such as Intel, IBM, AMD, and others are taking on the multicore road, and the number of cores varies between a couple cores to several tens of cores in the foreseeable future. Multicore processors typically share resources among the cores, such as caches, memory controllers, off-chip bandwidth, memory banks,

This research is funded through the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013)/ERC grant agreement no. 259295. S. Eyerman is a postdoctoral fellow of the Research Foundation-Flanders (FWO).
Authors' addresses: K. Du Bois, S. Eyerman, and L. Eeckhout (corresponding author), ELIS Department, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium; email: lieven.eeckhout@elis.ugent.be.

DOI 10.1145/2400682.2400688 http://doi.acm.org/10.1145/2400682.2400688

etc. Resource sharing increases hardware utilization, adds flexibility for a processor to adapt to varying workload demands (e.g., a thread with a large working set can allocate a large fraction of the shared cache), and can improve performance (e.g., fast communication between cores through the on-chip interconnection network and shared cache).

Resource sharing also comes with a significant drawback: coexecuting hardware threads affect each other's performance. For example, a thread allocating a large fraction of the shared cache may introduce additional conflict misses for other threads; likewise, memory accesses by a thread may close open pages in memory, thereby increasing memory access time for other threads. These inter-thread interferences may or may not have an effect on per-thread performance depending on whether they can be hidden by doing other useful work. As a result, resource sharing may affect per-thread performance in unpredictable ways. The key problem now is that system software (e.g., the OS or the VMM) assumes that all threads make equal progress, however, this is not the case when run on the underlying hardware: one thread may make faster progress than another one in the coscheduled workload mix, and the progress rates may be different across workload mixes across different timeslices. The mismatch between the assumption of equal progress by system software and the actual progress on the underlying hardware may make multicore processor scheduling ineffective. In other words, system software assumes that all threads make equal progress but in reality they do not. This may lead to undesirable properties at system level such as missed deadlines for (soft) real-time applications, non-satisfied service-level agreements, jitter on the response times for interactive workloads, unbalanced performance across coexecuting threads of a parallel workload, priority inversion, starvation, etc.

This article presents a per-thread cycle accounting architecture for multicore processors—to the best of our knowledge, this is the first comprehensive and hardware-efficient per-thread cycle accounting architecture for multicore processors that accounts for the major sources of inter-thread interference in shared resources. The accounting architecture estimates per-thread progress during multicore execution: for each thread, the counter architecture estimates what the execution time would be if run in isolation. Knowing per-thread progress enables system software to make multicore processor scheduling more effective. For example, the counter architecture can communicate to system software that one thread has made 5ms and the other thread has made 8ms of per-thread progress during a 10ms timeslice; this should enable system software to better understand how much progress each thread has made so far, and adjust the scheduling accordingly.

The proposed per-thread cycle accounting architecture addresses two major sources of inter-thread interference: (i) inter-thread misses or misses in the shared cache due to conflicts among the coexecuting threads, and (ii) waiting cycles, or lost cycles, due to resource and bandwidth contention in the memory subsystem, which prolong the memory access time of intra-thread misses (i.e., misses that also occur during isolated execution). The counter architecture accounts for hardware prefetching effects and also estimates what the impact is on overall performance, which is nontrivial given how out-of-order processor cores are designed to hide latencies. The counter architecture accounts for 74.3% of the interference cycles and estimates per-thread progress within 14.2% on average for an 8-core processor. The hardware cost is as small as 7.44KB for a multicore processor with eight cores and an 8MB last-level cache. This is a near 10$\times$ reduction in hardware cost compared to the pollution filter by Ebrahimi et al. [2010], while being 63.8% more accurate.

We demonstrate that per-thread cycle accounting improves multicore performance by making the scheduler progress aware, i.e., a progress-aware scheduler schedules threads that make slow progress more frequently. Our experimental results demonstrate the potential of per-thread cycle accounting: we report fairness improvements by 22.5% on average over progress-agnostic scheduling.

The remainder of this article is organized as follows. We first describe the various sources of inter-thread interference in multicore processors (Section 2) and detail on how the proposed counter architecture estimates the impact of the various sources of inter-thread interference on per-thread performance (Section 3). Sections 4 and 5 describe the experimental setup and the hardware cost of the proposed accounting architecture, respectively. We then evaluate the accuracy of the per-thread cycle accounting architecture (Section 6) and how it improves multicore scheduling (Section 7). Finally, we discuss related work (Section 8) and conclude (Section 9).

2. THREAD INTERFERENCE IN MULTICORE PROCESSORS

The first step towards designing a per-thread cycle accounting architecture is to understand the different sources of interference. Consider a multicore processor in which each core has private L1 instruction and data caches, and the last-level (L2) cache (LLC), the memory controller, the memory bus, and the main memory banks are shared. (We assume a shared L2 cache in this work, but the work can be trivially extended to shared L3 caches.) Throughout the article, we assume a single thread per core, hence we use the terms "thread" and "core" interchangeably. Upon a context switch on a core, system software (e.g., the OS) stores the interference counters (which we will detail later in the article) for the old thread, and reloads all counters for the new thread. This enables per-*thread* accounting, while the accounting hardware measures per-*core* interference.

2.1. Sources of Interference

Coexecuting threads on a multicore processor interfere with each other in each of the shared resources, causing per-thread performance to deteriorate compared to isolated execution. Each shared resource leads to different interference effects, which we discuss now.

LLC. Sharing the last-level cache between threads leads to extra conflict misses due to threads evicting each other's data. We refer to these conflict misses as *inter-thread misses*. In contrast, we define *intra-thread misses* as misses that also occur during isolated execution. Inter-thread misses do not occur during isolated execution and hence, their performance impact is potentially detrimental to per-thread performance: these memory references would be serviced by the LLC in isolated execution but turn into long-latency memory accesses during multicore execution.

Interconnection network. The on-chip interconnection network connects the cores to the shared cache (and to each other). A request of one core can be delayed due to a request by another core. Conflicts in the interconnection network thus prolong both the LLC hit and miss latency compared to isolated execution. Prolonging the LLC hit time due to conflicts in the interconnection network is unlikely to significantly affect per-thread performance, because the LLC hit latency (even with the additional conflict latency) is small enough so that it is effectively hidden through out-of-order execution in a balanced design [Eyerman et al. 2009]. For LLC misses, the additional conflict latency may have a significant effect because the processor cannot make progress while handling the LLC miss because of its long latency.

Memory bus. As with the interconnection network, a memory request issued by a core can hold the bus, between the LLC and main memory, possibly delaying requests by other cores. This causes memory accesses to take longer, which may have a significant impact on performance.

Memory bank effects. Main memory typically consists of a number of memory banks: these banks can handle memory accesses in parallel, thereby enabling memory-level parallelism (MLP). However, each bank can handle one access at a time only. This implies that while a bank is busy processing an access of a core, no other requests to that bank from other cores can be serviced. This increases the memory access time for the other cores.

An additional effect occurs in case of an open-page policy. Consider the example in which a thread accesses the same page twice and there are no intervening memory accesses to another page in the same bank, i.e., the page is loaded in the row buffer and both accesses are serviced from the row buffer. Now, another thread may interfere and may initiate a memory access to that same bank (but a different page) between the two memory requests by the first thread. This memory access will cause the row buffer to be written back to the memory bank and a new page to be loaded in the row buffer. The second memory access by the first thread will now see a row miss (instead of a row hit) and will need to load the page again into the row buffer. In other words, this second memory access will see a longer latency during multicore execution than it would see during isolated execution.

2.2. Impact of Interference on Performance

Not only is it important to detect interference among coexecuting threads, we also need to estimate how interference affects multicore execution time relative to isolated execution time. To this end, we leverage on the insights provided by interval analysis [Eyerman et al. 2009]. Interval analysis is an intuitive, mechanistic performance model for superscalar out-of-order processors and provides insight with respect to how miss events affect performance. Interval analysis focuses on the dispatch stage of a processor—dispatch here refers to the process of entering instructions from the front-end pipeline to the reorder buffer and issue queues—and reveals that the penalty for a miss event is the time between dispatch stalling upon a miss event and dispatch resolving after the miss event has been serviced. For a long-latency load, e.g., LLC miss or D-TLB miss, this means that its penalty equals the number of cycles dispatch stalls because of a filled-up reorder buffer (ROB) while the load miss resides at the ROB head. Karkhanis and Smith [2002] studied the various sources of performance loss due to long-latency loads and they concluded that the reorder buffer filling up is more common relative to the issue queue filling up with load-dependent instructions and the processor running out of rename register. We model all three cases: we quantify the long-latency miss penalty as the number of cycles where a long-latency load miss blocks the head of a filled-up ROB or filled-up issue queue, or when the processor runs out of rename registers.

It follows from this insight that the penalty due to an inter-thread miss equals the number of cycles the load miss resides at the head of a full ROB. Likewise, interference effects that prolong the penalty of an intra-thread miss, e.g., interference effects in the on-chip interconnection network and memory bus, can be quantified as the number of additional cycles due to interference while a long-latency load miss blocks commit at the head of a full ROB. We will leverage on these insights in our design of the counter architecture.

2.3. Quantifying Interference

Before describing the accounting architecture in detail, we first quantify the impact of interference on per-thread performance, and we identify the contribution of different sources of interference. We define *interference* as the relative increase in execution time

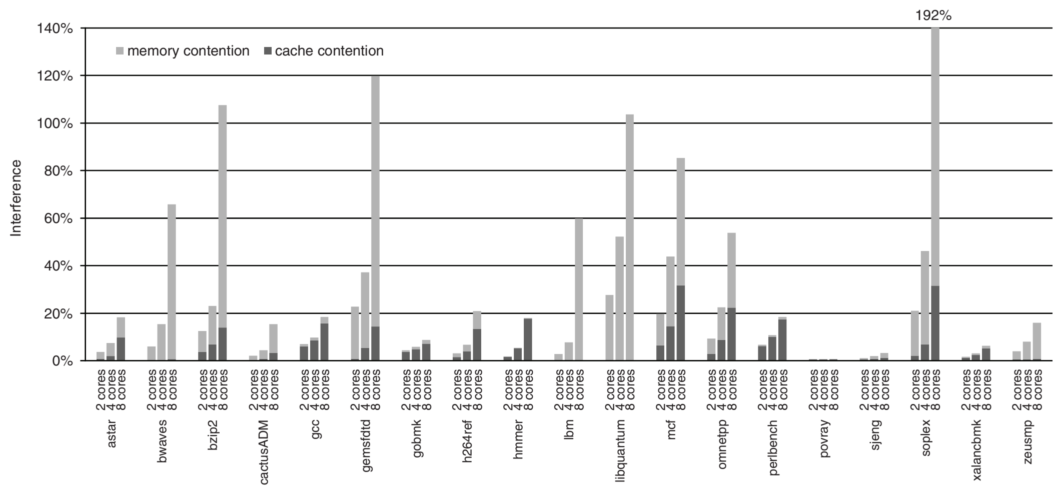


Fig. 1. Impact of inter-thread interference on per-thread performance for 2, 4, and 8 cores, breaking up interference in cache versus memory contention (average interference is reported across a set of job mixes per benchmark and assuming hardware prefetching).

between multicore and isolated execution.

$$Interference = \frac{T_{multicore} - T_{isolated}}{T_{isolated}} \tag{1}$$

Interference thus quantifies the increase in execution time on a multicore processor due to interference relative to isolated single-core execution. Through detailed simulation—see Section 4 for a description of the experimental setup—we find that interference is significant and that it increases with the number of cores: 9.3% on average for 2 cores, 19.5% for 4 cores, and 55.4% for 8 cores. The reason why interference increases with core count is that an increasing number of cores put increasingly more pressure on the shared resources, and hence, per-thread performance is affected more significantly.

To understand the relative contributions of the different sources of interference, Figure 1 makes a distinction between the interference due to inter-thread misses in the shared cache versus resource and bandwidth sharing in the memory subsystem (memory bus, memory banks, and open row policy). Some benchmarks seem to suffer more from cache sharing whereas other benchmarks suffer more from sharing in the memory subsystem. These interference numbers illustrate that the shared resources have substantial impact on per-thread performance, and by consequence, estimating interference is nontrivial (i.e., the null predictor would be highly inaccurate).

Figure 2 quantifies the impact of hardware prefetching on interference. The maximum interference level observed increases from 2.3× without prefetching to up to 3.8× with prefetching. The reason is that hardware prefetching puts even more pressure on the memory system's shared resources, which in its turn affects per-thread progress. In particular, a core that issues many prefetch requests may congest the memory subsystem and thereby degrade other cores' performance.

3. PER-THREAD CYCLE ACCOUNTING ARCHITECTURE

The central idea of per-thread cycle accounting is to implement an architecture in hardware that measures events which enable system software to estimate per-thread progress and act on it through scheduling to improve overall system performance, quality-of-service, etc. In other words, the counter architecture measures events such

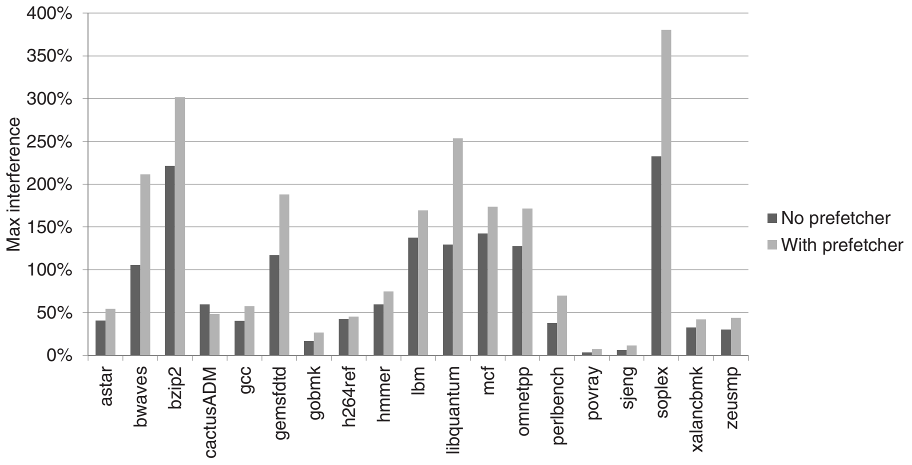


Fig. 2. The impact of prefetching on interference on an eight-core system (maximum interference is reported across 10 jobmixes per benchmark).

as the number of inter-thread misses and their penalty, as well as the number of waiting cycles for intra-thread misses, and communicates these events to system software at regular time intervals, e.g., at the end of each timeslice. System software then estimates per-thread progress and adjusts the scheduling to improve system performance.

The counter architecture makes a distinction between two sources of inter-thread interference in multicore processors: (i) inter-thread misses in the shared cache (i.e., the per-thread miss rate increases due to conflicts induced by other threads), and (ii) resource and bandwidth contention in the memory subsystem which causes intra-thread misses to take longer—we will refer to these additional cycles as *waiting cycles*. The following sections describe how the proposed counter architecture accounts for both sources of inter-thread interference.

3.1. Inter-thread Misses

We detect inter-thread misses using a structure called the Auxiliary Tag Directory (ATD). The inter-thread miss performance impact is then estimated through an accounting mechanism.

3.1.1. Auxiliary Tag Directory (ATD). The Auxiliary Tag Directory (ATD) is a structure private to each core that keeps track of what the status of the shared cache would be if it were private to that core; see also Figure 3. (The ATD was first proposed by Qureshi and Patt [2006] to keep track of the utility of the various ways in a shared cache to each of the cores; we use the ATD for a different purpose.) The ATD keeps track of the tags and replacement bits (not the data) due to accesses by the given core. An access to the shared cache accesses both the shared cache and the private ATD of that core. Only the shared cache returns data (from the cache itself if the access results in a hit, or from main memory if it is a miss), but both the shared cache and the ATD adjust the tag and replacement bits. An LLC miss is classified as an intra-thread miss if it also misses in the ATD; in case of a hit in the ATD, the LLC miss is classified as an inter-thread miss.

The ATD naturally handles positive cache interference for multi-threaded applications. Positive interference occurs when a thread brings data into the cache that is later accessed by other threads, and as a result, the other threads see a cache hit and not a

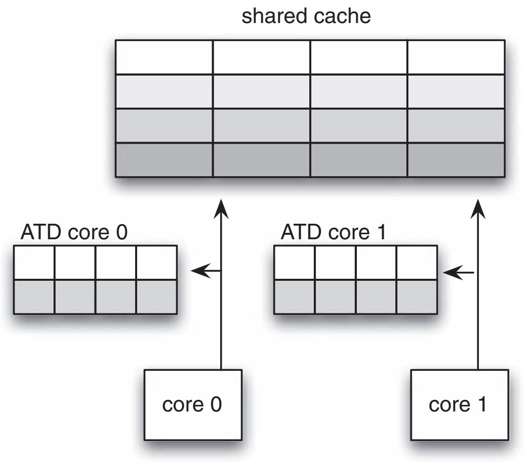


Fig. 3. The ATD samples a number of sets in the shared cache to identify inter-thread misses.

miss. This is called an *inter-thread hit* and is identified by the memory access being a hit in the LLC and a miss in the ATD.

In spite of the fact that the ATD contains only tag and replacement bits and no data, the hardware overhead is substantial: more than 6% of a 2MB shared cache *per core*, and thus the overhead for a multicore processor with a large number of cores quickly becomes substantial and practically infeasible. We therefore use set sampling [Kessler et al. 1994] to reduce the hardware overhead. We evaluated different sampling rates and found that sampling 32 out of 4096 sets yields the best balance between hardware overhead and accuracy; see the evaluation section in this article. The hardware overhead of the sampled ATD equals less than 0.05% of the shared cache per core.

3.1.2. Measuring the Performance Impact of Inter-thread Misses. Now that we know which misses in the shared cache are inter-thread versus intra-thread misses, the next question is to determine what their performance penalty is. We measure the number of interference cycles (miss penalty) due to inter-thread misses as the number of cycles an inter-thread miss blocks the head of a full ROB. The mechanism is as follows: as soon as the ROB is full and an inter-thread miss is at the ROB head, we start counting interference cycles. This requires being able to detect that the ROB is full, and a bit per ROB entry to keep track of whether a load miss is an inter-thread miss.

Because set sampling is employed in the ATD, as described above, we are unable to detect all inter-thread misses; we only know the status (inter-thread versus intra-thread miss) for those accesses that are sampled in the ATD. We therefore have to estimate the total number of interference cycles. We consider two approaches.

The first approach, the extrapolation approach, measures the penalty of the sampled inter-thread misses only, and then extrapolates to all inter-thread misses by multiplying by the sampling ratio.

$$\begin{aligned} \textit{inter-thread miss penalty} \approx\ & \textit{sampled inter-thread miss penalty} \\ & \times \frac{\textit{no. of cache accesses}}{\textit{no. of sampled cache accesses}}. \end{aligned} \tag{2}$$

The second approach, the interpolation approach, measures the penalty of all LLC misses (both inter-thread and intra-thread misses), and estimates the fraction of this penalty due to inter-thread misses by taking the ratio of the number of sampled inter-thread misses to the total number of sampled misses.

$$\textit{inter-thread miss penalty} \approx \textit{total miss penalty} \times \frac{\textit{no. of sampled inter-thread misses}}{\textit{no. of sampled misses}}. \quad (3)$$

We compare the accuracy of these approaches in the evaluation section. The conclusion is that the extrapolation approach is slightly more accurate, the reason being that the extrapolation approach measures the penalties of the sampled inter-thread misses, whereas the interpolation approach calculates the penalty for all misses (both inter-thread and intra-thread misses). The penalty for all misses may not be representative for the inter-thread misses, hence the extrapolation approach tends to be more accurate.

Computing the inter-thread miss penalty using the above formulas is relatively simple and can be done by system software. For example, at each timeslice, system software can read out the different counters and compute the inter-thread miss penalty. The counters can then be reset to profile the next timeslice. No additional hardware is needed for computing the above formulas.

3.2. Waiting Cycles

As discussed in Section 2, there are three main sources for waiting cycles or extra penalty that prolong the latency seen for intra-thread misses due to resource and bandwidth contention in the memory subsystem: memory bus conflicts, bank conflicts, and inter-thread row buffer misses. We now discuss how to count the number of waiting cycles and how to estimate their impact on overall performance.

3.2.1. Counting Waiting Cycles.

Bus contention. When a memory operation from one core occupies one of the buses (command, address, or data bus) while a memory operation from another core also wants to access the bus, then the latter incurs waiting cycles that would not have occurred in isolated execution. This is detected by inspecting the bus owner if a memory operation is ready to be scheduled on the bus and the bus is occupied. If the bus is owned by another core, then waiting cycles are accounted for.

Bank contention. An access that is delayed due to its destination bank being occupied by an access of another thread is detected similarly, and the extra waiting cycles are accounted for.

Inter-thread row buffer misses. This type of interference occurs when a row buffer hit in isolated execution becomes a row buffer miss during multicore execution, in case of an open-page memory policy. This occurs when a memory operation of another core accesses another row between the two consecutive accesses of one core to the same row. Since row buffer misses take considerably more time to be serviced than row buffer hits, this introduces an additional penalty and should be accounted for as waiting cycles.

Inter-thread row buffer misses are detected by maintaining an Open Row Array (ORA) per core (e.g., in the memory controller); see Figure 4. The ORA keeps track of the ID of the most recently accessed row per memory bank per core. If a row buffer miss hits in the private ORA, then the miss is caused by interference, and the extra penalty (i.e., the difference between the closed and open-page access times) is accounted for as waiting cycles. In case of a closed-page policy, the ORA is not needed.

Hardware prefetching. A long-latency load miss that also appears in the hardware prefetch queue (i.e., the prefetch is to be issued and/or is underway) is accounted waiting cycles if the load blocks commit at the head of a full ROB, as we will describe in the next

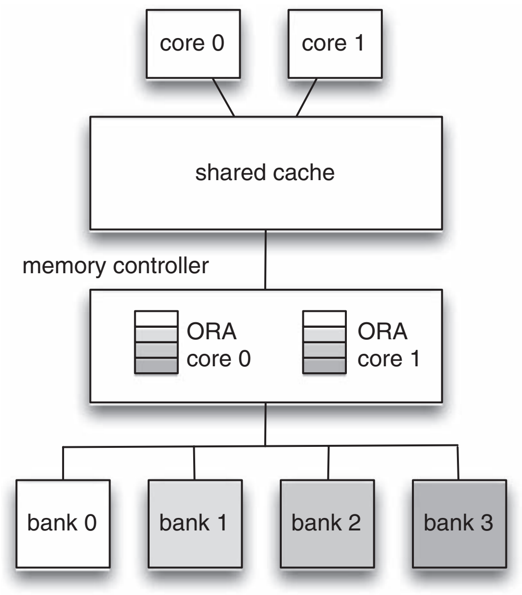


Fig. 4. The ORA keeps track of the most recently accessed row per memory bank per core.

section. This strategy basically assumes that the prefetch would be timely in isolated execution, while it is not during multicore execution because of contention. Although this is not always the case, this assumption keeps the accounting architecture simple and we found it to account for a major fraction of the interference due to prefetching.

Waiting cycles are kept track of in the MSHRs. The waiting cycles need to be kept track of for each individual memory access because there may be multiple outstanding inter-thread misses (whose latency is potentially hidden). The counter architecture keeps track of the waiting cycles in the MSHRs: we add a waiting cycle counter (10 bits) to each MSHR entry, and we add all waiting cycles pertaining to this memory access to this counter.

3.2.2. Estimating Impact Waiting Cycles on Performance. To estimate the performance impact of the waiting cycles, we use the insights provided by interval analysis [Eyerman et al. 2009]: a long-latency load miss only has impact on overall performance if it blocks the head of the ROB and causes the ROB to fill up. This implies that only waiting cycles need to be accounted as interference cycles if the long-latency load miss makes it to the ROB head and fills up the ROB. Based on this insight, we propose the following mechanism: if a miss blocks the head of the ROB and causes the ROB to fill up, then we add the miss' waiting cycles (that are kept track of in the MSHRs) to the per-core interference cycle counter.

We need to account for waiting cycles for the intra-thread misses only; the additional penalty incurred by inter-thread misses is accounted for as described before. The number of intra-thread misses is not readily available though. Instead we need to extrapolate on the sampled sets in the ATD. In line with what we described in Section 3.1.2, we consider two approaches. (Recall that all these calculations are done by system software; no additional hardware is needed.)

The extrapolation approach measures the waiting cycles of the sampled intra-thread misses and then extrapolates by multiplying with the ratio of cache accesses versus

Table I. Simulated Multicore Processor Configurations

Core frequency	2GHz
Core pipeline width	fetch: 8; dispatch, issue and commit: 4
ROB size	128 entries
Load and store buffer	96 entries each
Branch predictor	64K entry tournament, 4K entry BTB
L1 I-cache	32KB, 4-way, 64B line, 1 cycle
L1 D-cache	64KB, 4-way, 64B line, 2 cycles
Shared L2 cache	2MB (2 cores); 4MB (4 cores); 8MB (8 cores)
	8-way, 64B line, 10 cycles, 32 MSHRs
On-chip bus (L1 $\leftrightarrow$ L2)	2GHz, 32 byte
Hardware prefetcher	strided, per-core, degree of 4
Memory controller	FCFS, 16-entry write buffer
Memory bus	1333MHz, 64 bit
DRAM	667MHz DDR, 8 banks, 4KB row buffer
DRAM timing	9-9-9-7 (tRP-tRCD-CL-CWL)

the number of sampled accesses:

$$\begin{aligned} \textit{total waiting cycles} &\approx \textit{waiting cycles sampled intra-thread misses} \\ &\quad \times \frac{\textit{no. of cache accesses}}{\textit{no. of sampled cache accesses}}. \end{aligned} \tag{4}$$

The interpolation approach takes the number of waiting cycles of all misses and multiplies that with the estimated fraction of intra-thread misses. This approximates the number of waiting cycles for all intra-thread misses.

$$\begin{aligned} \textit{total waiting cycles} &\approx \textit{waiting cycles for all misses} \\ &\quad \times \frac{\textit{no. of sampled intra-thread misses}}{\textit{no. of sampled misses}}. \end{aligned} \tag{5}$$

For the same reasons as the ones discussed in Section 3.1.2, we find that extrapolation is more accurate than interpolation.

4. EXPERIMENTAL SETUP

4.1. Processor Configurations and Benchmarks

We use the gem5 simulator [Binkert et al. 2006] and simulate multicore processors with up to 8 cores. The L1 instruction and data caches are private to each core, and the L2 cache (LLC) is shared, and we assume an LLC of 2MB, 4MB, and 8MB for 2, 4, and 8 cores, respectively. We consider an aggressive stride-based hardware prefetcher between L2 and main memory. More details on the simulated processor configurations are available in Table I.

We consider the 19 SPEC CPU2006 benchmarks that properly run in our simulation environment. We use SimPoint [Sherwood et al. 2002] to select 1B-instruction representative samples from which we create a large number of multiprogram workloads. We consider all possible two-program combinations (190 two-program workloads in total[1]). For the four-program and eight-program workloads, we construct 10 three-program and 10 seven-program combinations, respectively, and we run all programs with each of these 10 multiprogram combinations. Hence, there are 190 four-program

[1]This is the number of combinations with repetition of 2 elements out of 19 elements, computed as $\binom{19+2-1}{2} = 190$.

and 190 eight-program workloads in total. All of these multiprogram workloads represent a good mix of memory-intensive and CPU-intensive benchmarks.

4.2. Error Metric and Simulation Procedure

Since the goal of the counter architecture is to estimate the execution time had the program run in isolation, we define error as the relative difference between the estimated and the actual isolated execution times.

$$\begin{aligned} Error &= \frac{T_{isolated,estimated} - T_{isolated,measured}}{T_{isolated,measured}} && (6)\\ &= \frac{(T_{multicore} - T_{estimated\ interference}) - T_{isolated,measured}}{T_{isolated,measured}}. && (7)\end{aligned}$$

A positive error implies an overestimation of the predicted isolated execution time or an underestimation of inter-thread interference; a negative error implies an underestimation of the predicted isolated execution time. We adopt the following simulation approach for computing the error. We first simulate a multi-program workload and stop the simulation when one of the programs has executed 1B instructions. We estimate the isolated execution times for each of the programs using the proposed counter architecture, i.e., $T_{isolated,estimated}$. We also determine the number of instructions executed for each program in the workload mix; one program has executed 1B instructions, the other programs have executed less than 1B instructions. For each program in the workload mix, we then run a single-threaded simulation for as many instructions as during multicore execution, and we determine the isolated execution time, i.e., $T_{isolated,measured}$. This procedure guarantees that the same amount of work is done during multicore execution as during isolated single-threaded execution for each program in the workload mix.

When reporting the average per-benchmark prediction error, we take the average of the absolute errors across all the workload mixes that include the benchmark. In particular, for the two-program workloads, we report the average absolute prediction error for a given benchmark across all the 19 two-program workload mixes; likewise, for the four-program and eight-program workloads, we report the average absolute prediction error per benchmark across all 10 four-program and all 10 eight-program workloads, respectively.

5. COUNTER ARCHITECTURE HARDWARE COST

Before evaluating the accuracy of the proposed counter architecture, we first quantify its hardware cost. The total hardware cost equals 952 bytes per core:

—ATD: 32 (sampled sets) $\times$ 8 (associativity) $\times$ 27 bit (tag + replacement) = 864 byte;
—marking (inter-thread) misses: 1 bit per ROB entry = 128 bit = 16 byte;
—the inter-thread latency counters in the MSHRs: 32 $\times$ 10 bit = 40 byte;
—ORA: 20 bit $\times$ 8 (banks) = 20 byte;
—the counters for the total number of accesses, the number of sampled accesses, and the number of sampled inter-thread misses: 3 $\times$ 20 bit = 8 byte;
—the total interference cycle counter: 32 bit = 4 byte.

The hardware cost for the proposed counter architecture is small (952 bytes per core), and scales linearly with the number of cores. Even for a multicore processor with a large number of cores, the total hardware cost is limited compared to the total transistor budget.

Not only is the amount of storage needed small, implementing the counter architecture should also be feasible in practice. The circuitry is localized to specific regions of the processor core. In particular, there is a set of counters in the ROB and MSHRs;

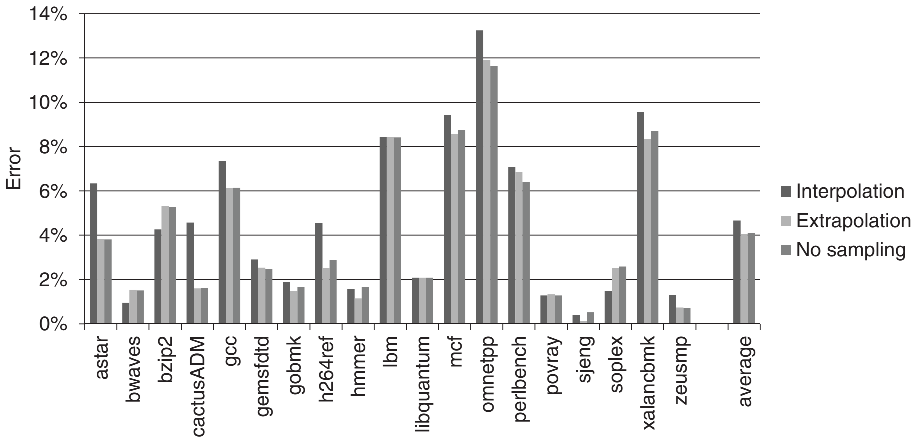


Fig. 5. Average isolated execution time estimation error per benchmark for Eqs. (2) and (3) (eight cores, fixed memory latency, 32 sampled sets), compared to no sampling.

there is the ATDs; and there is the ORAs. The counter architecture is unlikely to affect cycle time. The counters are read out by system software at regular (but coarse-grain) intervals, e.g., at the end of each timeslice. System software then uses these counts to predict per-thread progress.

6. EVALUATION

We evaluate the accuracy of the counter architecture in three steps. We first consider an idealized memory system and evaluate the counter architecture's accuracy for estimating the impact of inter-thread misses (additional misses in the shared cache) on overall performance. Second, we evaluate the counter architecture's overall accuracy while considering both inter-thread misses as well as waiting cycles on intra-thread misses; we consider a realistic memory system including hardware prefetching, memory banks, and an open-page policy. Finally, we compare our approach to the pollution filter which was recently proposed by Ebrahimi et al. [2010] for detecting inter-thread misses.

6.1. Estimating the Impact of Inter-thread Misses

We first evaluate the counter architecture's accuracy for estimating the effect of inter-thread misses on overall performance. To this end, we consider an idealized memory system in order to focus on inter-thread misses and eliminate the effect of waiting cycles on intra-thread misses. We assume a fixed memory access latency (100ns) and assume there are no bank conflicts. Figure 5 shows the isolated execution time error when using the extrapolation and interpolation approaches (Eqs. (2) and (3), respectively). The extrapolation approach slightly outperforms the interpolation approach with an average error of 4.05% versus 4.66%, respectively.

Figure 6 shows the impact of sampling frequency in the ATD on accuracy for both approximations. The error of the interpolation approximation seems to be less sensitive to sampling frequency compared to the extrapolation approach. The reason is that the interpolation approach measures the penalty of all misses to infer the penalty for the inter-thread misses, whereas the extrapolation approach measures the penalty of the sampled inter-thread misses only and then extrapolates to all inter-thread misses. At a low sampling frequency, the penalty for the sampled inter-thread misses is not

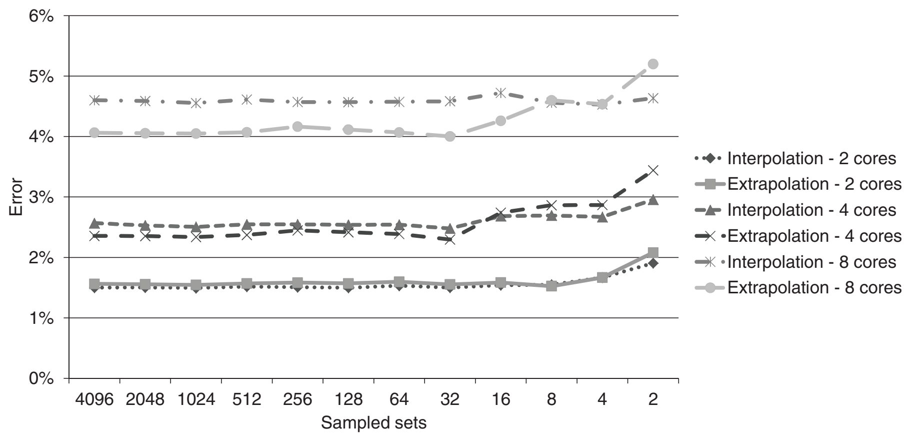


Fig. 6. Average isolated execution time estimation error for the interpolation and extrapolation approaches as a function of the number of sampled sets; we assume fixed memory access latency.

representative for the other inter-thread misses, hence accuracy degrades. We find that 32 sampled sets is a good design point for two, four, and eight cores.

6.2. Overall Accuracy Evaluation

In the previous section, we assumed idealized memory (fixed access latency) in order to evaluate the counter architecture's accuracy with respect to estimating the impact of inter-thread misses. We now consider a realistic memory system with multiple banks along with an open-page policy; further, we assume hardware prefetching is enabled. This allows for evaluating the accounting architecture's ability to accurately estimate the effect of both inter-thread misses and waiting cycles. Figure 7 shows the measured versus estimated interference for two, four, and eight cores. The counter architecture achieves an average (absolute) error of 3.75% for 2 cores, 5.57% for 4 cores, and 14.2% for 8 cores. This is fairly accurate given the level of interference, which equals 9.3% for 2 cores, 19.5% for 4 cores, and 55.4% for 8 cores; in other words, the accounting architecture captures 74.3% of the inter-thread interference on average for 8 cores. Note that although the absolute error increases with increasing core count, e.g., it increases from 5.57% for 4 cores to 14.2% for 8 cores, the relative error compared to the level of interference actually decreases from 28.5% for 4 cores to 25.6% for 8 cores. In other words, absolute error increases with core count but so does the level of interference, hence in the end, the relative accuracy of the counter architecture is fairly stable and actually decreases with core count, i.e., the proposed counter architecture is able to consistently capture the most significant sources of inter-thread interference. Further, we expect absolute accuracy to improve given microarchitecture enhancements that reduce inter-thread interference, such as a advanced network-on-chip topologies, multiple memory controllers, limiting the number of cores that share a cache, etc.

The results shown so far presented error numbers that are averaged across a number of multiprogram workloads, e.g., there are 19 two-program workloads and 10 four-program and 10 eight-program workloads per benchmark. Figure 8 shows the same data but does not average out across a number of multi-program workloads; i.e., there is a data point for each workload (190 in total). The graph shows a cumulative distribution for the interference and the error: the horizontal axis shows the fraction of workloads for which the interference and error are below the corresponding value on the vertical axis. This graph shows the amount of variation in both the interference and the error. We observe that interference can be very high for some workloads, up to

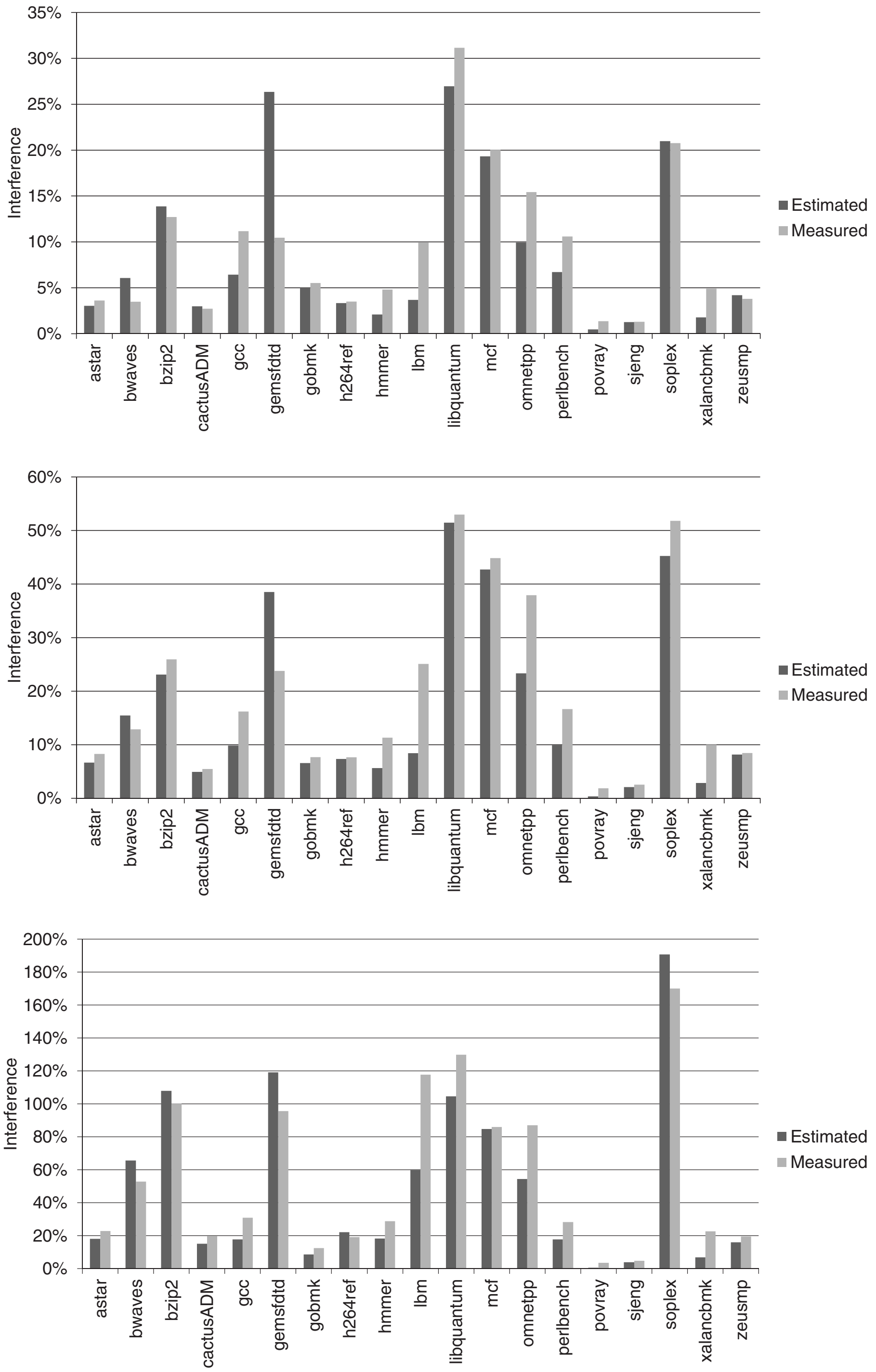


Fig. 7. Estimated versus measured interference for (a) a dual-core, (b) a quad-core, and (c) an eight-core system.

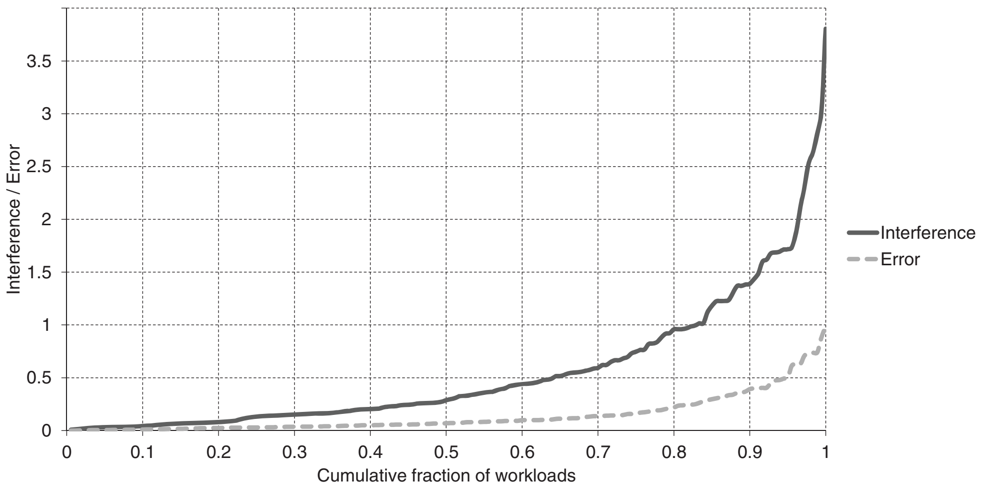


Fig. 8. Interference and error for estimating isolated execution time for an 8-core processor; workloads are sorted along the horizontal axis.

3.8×. The proposed cycle accounting architecture accurately identifies the workloads with high interference levels, and the error is substantially lower compared to the level of interference. In particular, for 80% of the workloads, interference is as large as 92%, yet the error is less than 21%; similarly, for 70% of workloads, interference is as large as 59% with an error below 13.6%. This graph shows once more that the accounting architecture is able to measure a large fraction of the interference in all cases: a higher interference results in a higher absolute error, but the relative error remains approximately the same (around 25%).

6.3. Error Analysis

In order to better understand the different sources of error, we set up a number of experiments and we quantified how the error is affected by various sources of interference for the various benchmarks; see Figure 9. For the first bar, we assumed no bank nor row buffer conflicts (fixed memory access latency), no hardware prefetching and full (i.e., nonsampled) ATDs. In this case, the accounting architecture achieves an average absolute error of 4.1% on an 8-core system. This error follows mainly from second-order effects that are not captured. For example, intra-thread waiting cycles hidden underneath inter-thread misses during multicore execution are possibly not hidden in isolated execution; in this case, the accounting architecture would account for the waiting cycles, although it should not. We chose not to consider second-order effects in order not to complicate the accounting architecture hardware.

The second bar shows the impact of sampling only 32 sets of the cache. This has no noticeable effect on accuracy. Adding banks and considering an open-page policy (third bar) increases the average error to 11.5%, and adding prefetchers (fourth bar) increases the overall error to 14.2%. These errors also stem from second-order effects that are not modeled in our cycle-accounting method, to reduce the overhead and complexity of the hardware additions. For example, when an inter-thread miss causes waiting cycles (e.g., due to a bank conflict) for an intra-thread miss of the same thread, these cycles are not accounted since both misses belong to the same thread. Accounting for this would require a categorization of every miss into inter- or intra-thread, which is impossible using a sampled ATD, and would also need communicating the inter- and intra-thread miss information to the memory controller, which complicates the design.

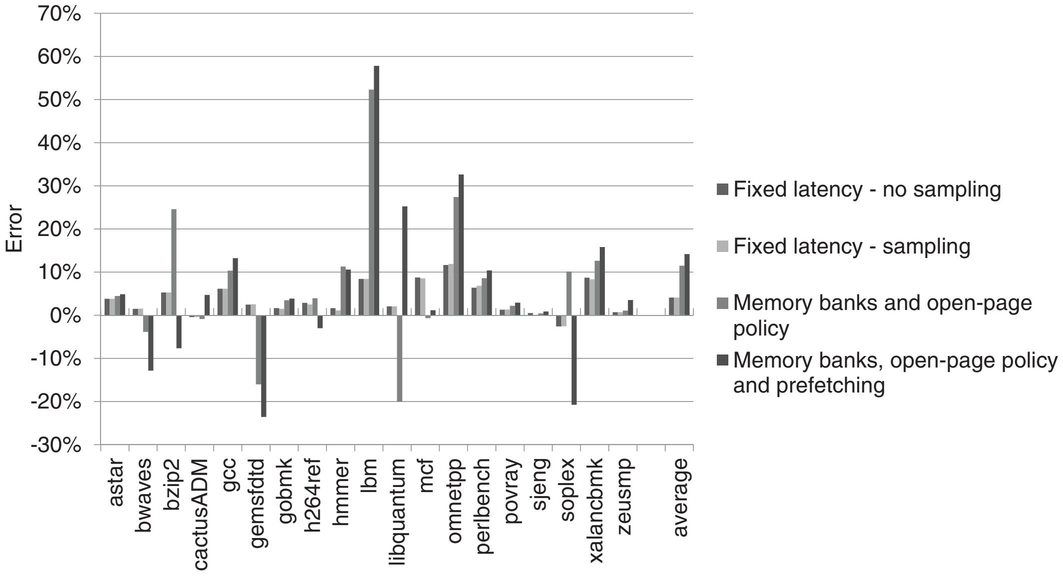


Fig. 9. Error analysis per benchmark for an 8-core processor.

Furthermore, Figure 9 also shows that the resulting error can be positive or negative (we took the absolute values to calculate the averages, such that these errors do not compensate for each other). This shows that our technique is not biased, i.e., there is no consistent under- or overestimation. As a result, additional accounting to decrease the error in the case interference is overestimated tends to also increase the error in case the interference is underestimated and vice versa. This means that in order to reduce the error consistently, both underestimation and overestimation cases need to be handled, which would make the accounting architecture overly complex.

6.4. Comparison against the Pollution Filter

Ebrahimi et al. [2010] present an alternative mechanism for detecting inter-thread misses using a structure called the *pollution filter*. The pollution filter consists of an array of bits and there are as many bits in the pollution filter as there are sets in the shared cache; and there is one pollution filter per core. A bit indicates whether a block was evicted from the cache by another thread or not. The bits are indexed by the cache block address, alike how the cache is indexed. The pollution filter works as follows. If a replacement in a cache set causes the eviction of a cache block most recently accessed by another core, then the corresponding bit in the pollution filter of the previously referencing core is set. If a miss occurs, the pollution filter is consulted, and if the bit is set, the miss is classified as an inter-thread miss. This indicates a scenario in which the cache access would have been a hit in isolated execution but is a miss in multicore execution. When the data of the inter-thread miss is fetched from memory and inserted into the cache, the corresponding bit is reset in the pollution filter.

The advantage of a pollution filter over a sampled ATD is that the pollution filter can make a classification for every miss in terms of whether the miss is an inter-thread versus intra-thread miss. In other words, the pollution filter does not suffer from sampling inaccuracy. On the other hand, since multiple cache blocks map to the same bit, a pollution filter is subject to aliasing. This is not the case for the sampled sets in the ATD.

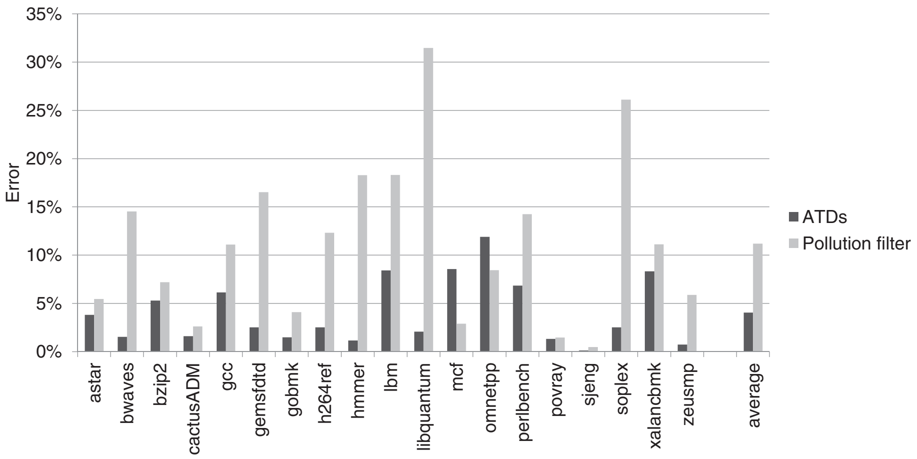


Fig. 10. Average isolated execution time estimation error per benchmark for the ATD (32 sampled sets and 6.75KB total hardware cost) versus the pollution filter (16K entries and 64KB total hardware cost); we assume eight cores and a fixed memory access latency.

The hardware cost of a pollution filter is the number of sets times one bit; and there is a pollution filter per core. Additionally, a thread ID has to be kept track of for each cache block in the shared cache: this is to detect whether or not the evicted block was most recently accessed by the same thread. The ATD approach, which we advocate, does not need to keep track of thread IDs in the cache blocks. Figure 10 compares the accuracy of the ATD versus the pollution filter assuming 8 cores and an 8MB LLC. The hardware cost for the sampled ATDs equals 8×864 bytes or 6.75KB in total. The pollution filter, on the other hand, requires 64KB for 8 cores (and 8MB L2): 48KB for keeping track of the thread IDs in the shared cache plus 8 times 2KB for the pollution filters. In other words, the sampled ATDs involve approximately $10\times$ less hardware than the pollution filter. Figure 10 shows that the sampled ATD is substantially more accurate than the pollution filter, in spite of its smaller hardware budget: 4.05% average error for the ATD versus 11.2% for the pollution filter—a 63.8% improvement in accuracy while being an order of magnitude more efficient in hardware cost. The interesting observation is that accurately measuring the miss penalty for a limited number of inter-thread misses leads to more accurate predictions as opposed to determining all the inter-thread misses (and being limited by aliasing effects at the set level).

7. MULTICORE SCHEDULING

Now that we have evaluated the counter architecture's accuracy, the question is how this translates into multicore performance. Progress-aware scheduling leverages the per-thread cycle accounting architecture to track per-thread progress and schedules slowly progressing threads more frequently so that they are able to catch up and achieve better performance. Progress-agnostic scheduling assumes that each thread makes equal progress during each timeslice, however, a thread that suffers more from resource contention will observe a higher slowdown compared to other threads. The pitfall is that the scheduler is unaware of this slowdown, which may lead to severely degraded performance for workloads that suffer significantly from resource contention.

To evaluate thread-progress-aware scheduling, we set up the following experiment. We consider the 4-program and 8-program workload mixes which we schedule on a 2-core and 4-core system, respectively (see Table I for the configuration of these systems); we assume a 5ms timeslice in these experiments, and we simulate until at least one benchmark has executed 1 billion instructions[2]. The scheduling techniques are implemented in the simulator itself, as we do not simulate the operating system.

The baseline progress-agnostic scheduling policy schedules programs such that they all get an equal amount of timeslices; e.g., round-robin achieves this property. Progress-aware scheduling, on the other hand, tracks progress for each of the programs in the mix, and prioritizes the programs with the highest current slowdown to be scheduled first. Slowdown is computed as the execution time on the multicore system divided by the estimated isolated execution time. In other words, progress-aware scheduling aims at speeding up slow-progress programs so that all users experience good performance and none of the programs experiences huge slowdowns nor starvation.

Figure 11 reports the progress-agnostic and progress-aware scheduling fairness observed for each of the workload mixes for the 2-core and 4-core systems. Fairness [Eyerman and Eeckhout 2008] is defined as the progress of the slowest program divided by the progress of the fastest program in the job mix. A fairness of 1 means that each thread has made the same progress, while a zero fairness means that at least one thread is starving. Each point represents the progress-agnostic (Y-axis) and progress-aware (X-axis) scheduling fairness of a specific workload. Points on the indicated bisector have the same fairness for both scheduling policies. Points beneath the bisector have higher fairness for the progress-aware scheduling, while points above the bisector show a smaller fairness for progress-aware scheduling compared to progress-agnostic scheduling.

Progress-aware scheduling improves fairness substantially over progress-agnostic scheduling for most of the workloads: the majority of the points are located beneath the bisector. For the points at the bottom right corner, the fairness improvement is the largest: a smaller than 0.1 fairness for the progress-agnostic scheduling becomes a bigger than 0.9 fairness for the progress-aware scheduling, which uses the proposed cycle-accounting architecture. We observe an average fairness improvement of 20.3% and 24.8% for 2 and 4 cores, respectively.

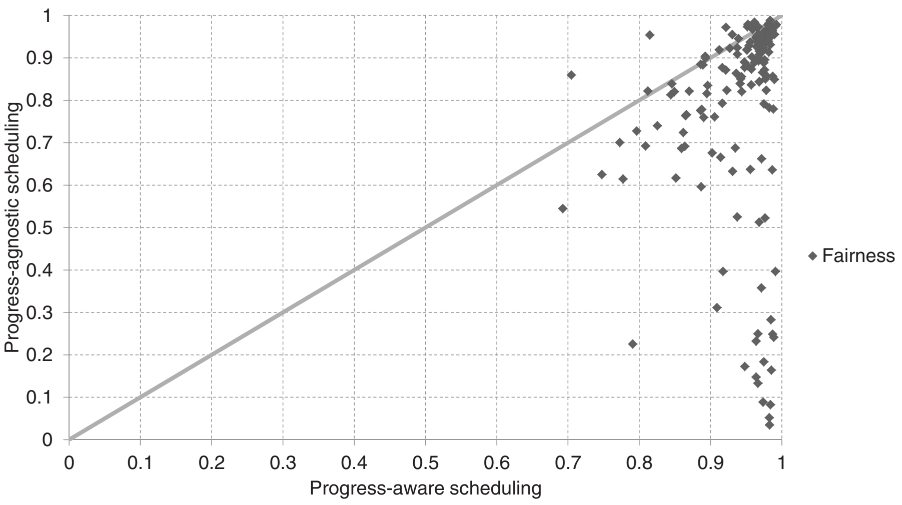
Improving fairness for threads running on a multicore processor is important for avoiding missed deadlines in soft real-time applications, for reducing jitter in the response time for interactive applications, for guaranteeing fairness in consolidated environments, for delivering service-level agreements, for balanced performance in parallel workloads, etc. We believe that per-thread cycle accounting is foundational for this set of applications, which is further supported by the experimental evaluation in this work.

8. RELATED WORK

8.1. Per-Thread Cycle Accounting

Eyerman and Eeckhout [2009] first presented the concept of per-thread cycle-accounting and they proposed a per-thread cycle, accounting architecture for Simultaneous MultiThreading (SMT) processors. The per-thread cycle-accounting architecture estimates the per-thread progress for each thread in an SMT processor. An SMT processor shares almost all of its core-level resources among the coexecuting threads (e.g., ROB entries, functional units, rename registers, L1 caches, branch predictor, etc.). The focus of this prior work in per-thread cycle accounting has been on estimating the impact of core-level resource sharing, and limited focus was put towards estimating the impact of resource sharing in the memory hierarchy. This article adds memory system interference modeling and improves cache sharing interference modeling—interpolation versus extrapolation. The previously proposed

[2]We were unable to run 16-program workload mixes on an 8-core system because of physical memory constraints on our simulation host machine.



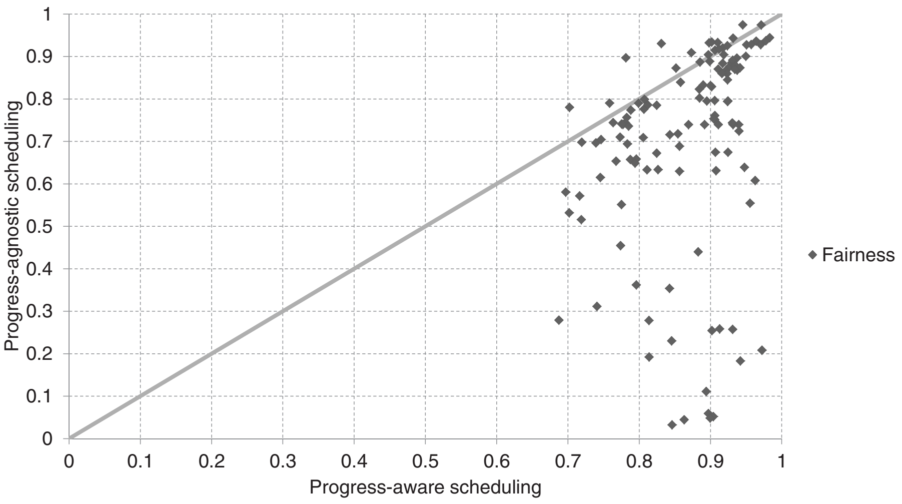


Fig. 11. Fairness results for progress-aware and progress-agnostic scheduling for (a) 4-program mixes on 2 cores, and (b) 8-program mixes on 4 cores.

SMT accounting architecture is orthogonal to the one proposed in this article and can thus be integrated in case of a multicore of SMT cores.

8.2. Quantifying Impact of Cache Sharing

A couple recent papers account for the impact of cache sharing, but do not model the impact of memory bus and main memory bank sharing. Luque et al. [2009] propose a scheme to assess the impact of inter-thread cache misses. A limitation of this mechanism is that it needs to know for every miss whether it is an inter-thread miss or intra-thread miss, which incurs significant hardware overhead. Zhou et al. [2009] propose a complex set of counters to estimate the performance impact of inter-thread misses. Its accuracy quickly drops when sampling is employed though. The cycle, accounting architecture proposed in this article is more hardware efficient and in addition captures the impact of both cache and memory sharing.

8.3. Quantifying Impact of Memory System Sharing

Mutlu and Moscibroda [2007] focus on contention in main memory and they propose a mechanism to quantify the impact of bus and memory bank sharing on per-thread performance. A fairly complicated scheme is needed for counting the number of pending events; in addition, an empirical parameter is needed to fine-tune the proposed technique. Our scheme is simpler and does not involve empirical components. Further, this work focuses on main memory sharing only, and does not address cache sharing. Finally, the authors do not evaluate the accuracy of the proposed scheme for predicting the impact of memory system sharing on per-thread performance; instead, the scheme is evaluated indirectly through a mechanism to improve fairness in CMPs.

Ebrahimi et al. [2010] proposed a mechanism for estimating the performance impact of both cache and memory sharing. They propose the pollution filter to detect inter-thread misses, which we have shown to be less hardware efficient and less accurate than sampled ATDs; see Section 6.4. Further, Ebrahimi et al. evaluate the accuracy of the proposed mechanism indirectly only (by showing that it can improve fairness in a multicore memory system) and they do not report per-thread progress estimates. Finally, their scheme handles overlap effects by using a single bit per core that indicates whether the core is experiencing an interference event (multiple interference events add only one unit per cycle); the mechanism then counts the number of cycles this bit is set in order to know the total number of interference cycles. This approach implies that (potentially long) wires need to be provided from the memory system to this per-core bit. In contrast, our mechanism measures the various interference events locally, and the resulting performance impact is calculated using formulas that are evaluated in system software that take the different counts as an input.

Ebrahimi et al. [2011] present a mechanism that controls prefetch behavior to reduce inter-thread interference caused by prefetch requests. In our proposal, we measure the impact of inter-thread interference, including prefetch requests, which is orthogonal to their policy. However, they also propose to measure the delay a prefetch causes on a demand miss. Our mechanism does this in a natural way, since only demand misses can block a processor core, and once scheduled, prefetch requests are considered as normal reads. Furthermore, we also account for the case where prefetch requests are delayed by demand misses and prefetch requests by other threads, which is not covered by Ebrahimi et al.

8.4. Cache Partitioning

A large body of recent work has focused on cache partitioning in multicore processors, see for example; Iyer [2004], Iyer et al. [2007], Jaleel et al. [2008], Kim et al. [2004], Nesbit et al. [2007], Qureshi and Patt [2006], and Zhou et al. [2009]. These proposals did not quantify per-thread progress, but aimed at improving multicore throughput while guaranteeing some level of fairness among coexecuting jobs. Guo et al. [2007] propose a framework to provide QoS in CMP systems. To accomplish this, they make use of dynamic cache partitioning. As in Qureshi and Patt [2006], they use sampled ATDs to determine the impact of assigning a limited number of sets to an application. They do not model interference in main memory though. Our mechanism measures interference in all of these components.

8.5. Thread Criticality

Bhattacharjee and Martonosi [2009] predict critical threads, or threads that run slower than other threads in parallel workloads. The motivation is to give more resources to critical threads so that they run faster, offload tasks from critical threads to noncritical threads to improve load balancing, etc. They determine thread criticality by tracking and weighting the number of cache misses at different levels in the memory hierarchy.

9. CONCLUSION

Resource sharing leads to interference among coexecuting threads in multicore processors because of contention effects: coexecuting threads compete for the shared resources, such as caches, off-chip bandwidth, memory banks, etc., which leads to unpredictable per-thread performance. The fundamental problem now is that system software assumes that all coexecuting threads make equal progress although this is not guaranteed by the underlying hardware. This may lead to various detrimental effects at the system level such as reduced quality-of-service, nonsatisfied service-level agreements, unbalanced parallel execution, priority inversion, etc., which severely complicates multicore processor scheduling.

This article presented a hardware-efficient per-thread cycle-accounting architecture that tracks per-thread progress during multicore execution. The cycle-accounting architecture estimates the impact of inter-thread misses in the shared cache as well as the resource and bandwidth sharing in the memory subsystem, including the memory bus, bank conflicts, and row buffer conflicts. The accounting architecture estimates the impact of resource sharing on per-thread performance and reports to system software how much isolated progress each thread has made during multicore execution. We report average error numbers of 14.2% for an 8-core processor—the accounting architecture captures 74.3% of the inter-thread interference on average. Hardware cost is limited to 7.44KB for an eight-core system, which is almost 10× smaller than prior work, while being 63.8% more accurate. System software, when made aware of per-thread progress, improves multicore fairness by scheduling slowly progressing threads more frequently: we report an average fairness improvement of 22.5% over progress-agnostic scheduling.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive and insightful feedback.

REFERENCES

BHATTACHARJEE, A. AND MARTONOSI, M. 2009. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 290–301.

BINKERT, N. L., DRESLINSKI, R. G., HSU, L. R., LIM, K. T., SAIDI, A. G., AND REINHARDT, S. K. 2006. The M5 simulator: Modeling networked systems. *IEEE Micro 26,* 4, 52–60.

EBRAHIMI, E., LEE, C. J., MUTLU, O., AND PATT, Y. 2011. Prefetch-Aware shared-resource management for multi-core systems. In *Proceedings of the International Symposium on Computer Architecture (ISCA).*

EBRAHIMI, E., LEE, C. J., MUTLU, O., AND PATT, Y. N. 2010. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 335–346.

EYERMAN, S. AND EECKHOUT, L. 2008. System-Level performance metrics for multi-program workloads. *IEEE Micro 28,* 3, 42–53.

EYERMAN, S. AND EECKHOUT, L. 2009. Per-Thread cycle accounting in SMT processors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 133–144.

EYERMAN, S., EECKHOUT, L., KARKHANIS, T., AND SMITH, J. E. 2009. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst. 27,* 2.

GUO, F., SOLIHIN, Y., ZHAO, L., AND IYER, R. 2007. A framework for providing quality of service in chip multiprocessors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 343–355.

IYER, R. 2004. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *Proceedings of the International Conference on Supercomputing (ICS)*. 257–266.

IYER, R., ZHAO, L., AMD R. ILLIKKAL, F. G., MAKINENI, S., NEWELL, D., SOLIHIN, Y., HSU, L., AND REINHARDT, S. 2007. QoS policies and architecture for cache/memory in CMP platforms. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 25–36.

JALEEL, A., HASENPLAUGH, W., QURESHI, M., SEBOT, J., STEELY, JR., S., AND EMER, J. S. 2008. Adaptive insertion policies for managing shared caches. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 208–219.

KARKHANIS, T. AND SMITH, J. E. 2002. A day in the life of a data cache miss. In *Proceedings of the 2nd Annual Workshop on Memory Performance Issues (WMPI) held in conjunction with ISCA*.

KESSLER, R. E., HILL, M. D., AND WOOD, D. A. 1994. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Trans. Comput. 43,* 6, 664–675.

KIM, S., CHANDRA, D., AND SOLIHIN, Y. 2004. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 111–122.

LUQUE, C., MORETO, M., CAZORLA, F. J., GIOIOSA, R., BUYUKTOSUNOGLU, A., AND VALERO, M. 2009. ITCA: Inter-task conflict-aware CPU accounting for CMPs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 203–213.

MUTLU, O. AND MOSCIBRODA, T. 2007. Stall-Time fair memory access scheduling for chip multiprocessors. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 146–160.

NESBIT, K. J., LAUDON, J., AND SMITH, J. E. 2007. Virtual private caches. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 57–68.

QURESHI, M. K. AND PATT, Y. N. 2006. Utility-Based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 423–432.

SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 45–57.

ZHOU, X., CHEN, W., AND ZHENG, W. 2009. Cache sharing management for performance fairness in chip multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 384–393.

Received June 2012; revised September 2012; accepted September 2012