A Hybrid Hardware/Software Approach to Efficiently Determine Cache Coherence Bottlenecks *

Jaydeep Marathe[1] Frank Mueller[1] Bronis de Supinski[2]

[1] Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7534

[2] Lawrence Livermore National Laboratory
Center for Applied Scientific Computing
L-561, Livermore, CA 94551

e-mail: mueller@cs.ncsu.edu

Abstract

High-end computing increasingly relies on shared-memory multiprocessors (SMPs), such as clusters of SMPs, nodes of chip-multiprocessors (CMP) or large-scale single-system image (SSI) SMPs. In such systems, performance is often affected by the sharing pattern of data within applications and its impact on cache coherence. Sharing patterns that result in frequent invalidations followed by subsequent coherence misses create cache coherence bottlenecks with significant performance penalties. Past work on identifying coherence bottlenecks based on tracing memory accesses incurs considerable runtime overhead and does not scale well with increasing problem sizes, which makes it infeasible to use with real-world programs.

In this paper, we introduce a novel low-cost, hardware-assisted approach to determine coherence bottlenecks in shared-memory OpenMP applications. We assess the merits of our approach on a contemporary SMP platform. Specifically, we assess the feasibility of lossy tracing to pin-point coherence problems in applications. We evaluate the qualitative and quantitative trade-offs between tracing overhead and accuracy of the generated coherence traffic metrics, correlated to memory access points at the program source level.

Our lossy tracing mechanism closely approximates the degree of accuracy of determining coherence misses in full traces for most of the benchmarks we study while reducing run-time execution overhead and trace sizes by one to two orders of magnitude. To the best of our knowledge, this novel method significantly outperforms any of the prior approaches and, for the first time, makes cache coherence analysis feasible for long-running applications.

* This work was supported in part by NSF grants CAREER CCR-0237570, CNS-0406305, CCF-0429653 and through the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under subcontract # B540203. Part of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48, UCRL-CONF-212502.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profi t or commercial advantage and that copies bear this notice and the full citation on the fi rst page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specifi c permission and/or a fee.

ICS'05, June 20–22, 2005, Boston, Massachusetts, USA.
Copyright © 2005 ACM 1-?????-???-?/??/0006. . . $5.00.

Categories and Subject Descriptors B.8.2 [*Performance and Reliability*]: Performance Analysis and Design Aids; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Measurement, Performance

Keywords Hardware performance monitoring, dynamic binary rewriting, program instrumentation, cache analysis, SMPs, coherence protocols

1. Introduction

Recent advances in microprocessor design have been reflected in high-end computing platforms that increasingly rely on shared-memory multiprocessors (SMPs). Such platforms may be clusters of SMPs with multiple processing chips sharing memory over a bus-based protocol (*e.g.*, Intel Xeons), they may be nodes that rely on chip-multiprocessors (CMPs, *e.g.*, the IBM Power5), or even large-scale single-system image (SSI) SMPs (*e.g.*, the SGI Altix). Scientific codes on such systems often use multiple threads *via* POSIX Threads or OpenMP to solve a single problem by sharing data. The application's performance is often affected by the sharing pattern of data and its impact on cache coherence. In fact, sharing patterns that result in frequent invalidations followed by subsequent coherence misses represent cache coherence bottlenecks with significant performance penalties. Furthermore, hardware complexity makes it increasingly difficult for programmers to assess the effects on shared resources, specifically those imposed by cache coherence traffic between processors for the multitude of architectural variations (bus-based SMPs vs. CMPs vs. directory-based SSI SMPs). Hence, the effort of programming these platforms motivates a performance analysis methodology that aids users in detecting bottlenecks.

Prior work on cache coherence focused on simulation of coherence protocols and performance enhancements techniques to reduce coherence traffic. Architectural simulators support a multitude of coherence models in their implementation. These simulators and systems operate at different levels of abstraction ranging from cycle-accuracy over instruction-level [8, 12, 25, 4, 9] to the operating system interface [27]. Past work on the performance tuning concentrates on program analysis to derive optimized code [15, 29].

More recent work on identifying coherence bottlenecks is based on tracing memory accesses *via* dynamic binary rewriting [20]. This approach reduces the trace overhead by an order of a magnitude or more over conventional hardware simulators. But the approach still incurs considerable runtime overhead when compared to the un-instrumented performance of an application and does not

scale well with increasing problem sizes. While it may be useful for hot-spot analysis over short periods of time, it is infeasible for the analysis of an entire execution of a long-running application. This discourages programmers from using such analysis approaches in practice. The high overhead of past approaches results from the reliance on software-based techniques to obtain data traces, either by means of slow hardware simulations or *via* software instrumentation with significant overhead per access point.

We have developed a novel approach to identify coherence problems. Instead of a pure software solution, our hybrid hardware/software approach efficiently aids programmers in parallelize programs. More specifically, we introduce a unique low-cost, hardware-assisted approach to determine coherence bottlenecks in shared-memory OpenMP applications.

In this paper, we assess the merits of our approach on a contemporary SMP platform. We investigate the feasibility of *lossy tracing* to pin-point coherence problems in applications. We evaluate the qualitative and quantitative trade-offs between tracing overhead, degree of lossiness and the accuracy of the generated coherence metrics, correlated to memory access points at the program source level.

For the set of benchmarks studied, our lossy tracing method usually provides a high degree of accuracy with an order of magnitude savings in execution overhead and storage requirements for traces. This makes our approach attractive for main-stream tool support.

Our work provides insight into a synergistic approach between software tools and hardware monitoring (together with requirements on hardware performance monitoring) to uncover the potential to pin-point cache coherence problems. Benefits of this work extend from existing SMP architectures to small-scale CMPs and are likely to only increase with future large-scale CMPs on the horizon that will deliver an increasing potential for shared-memory multi-processing.

The paper is structured as follows. First, we demonstrate the usefulness of detailed source code-correlated coherence metrics. Then, we describe our hybrid hardware/software monitoring approach. Subsequently, we detail the experimental setup and present measurements for our experiments. Finally, we contrast our approach with prior work and summarize our contributions.

2. Source-Correlated Statistics

In prior work, we used a full-tracing approach to extract complete access traces from OpenMP applications [19]. These traces we fed to an incremental coherence simulator, which generated detailed source-code correlated coherence metric information. In this paper, we compare the accuracy of this simulator's results based on a new hardware-assisted lossy-tracing approach. Before detailing our new approach, let us motivate the need for source-code correlated coherence characteristics.

Consider SMG2000, a production-quality OpenMP benchmark from the ASCI Purple suite [1]. The example stems from our prior work [19]. SMG2000 is a large benchmark with approximately 24,000 lines of code in over 72 files and approximately 69 OpenMP regions. Using a conventional architecture simulator or hardware performance counters, coarse-level results can be obtained, similar to those shown in figure 1(a). The numbers indicate a possible coherence bottleneck (most L2 misses are coherence misses). But which parts of the source code are responsible for the bottleneck? What source code references compete for the same shared data causing invalidations and coherence misses?

Fundamentally, these questions cannot be answered using only aggregate metrics; we need an ability to "drill-down" and abstract the coherence metrics to elements in the high-level source code. Our coherence simulator generates such correlated results (shown in Figure 1(b)). Top references in processor-1 are suffering coherence misses and true/false sharing invalidations, depicted in descending order. This information provides insight into sharing patterns in the application and guides the programmer towards probable causes and optimization strategies. *E.g.*, the table shows that the rp_Read[] reference on line 289 of smg_residual.c suffered large amounts of coherence misses and false-sharing invalidations.

3. Hybrid Hardware/Software Monitoring

The problem of obtaining data traces is twofold. First, obtaining traces through software instrumentation (as in the last section) requires a modification of the application, either *via* static instrumentation through the compiler, a static binary rewriter or *via* dynamic binary rewriting. In either case, additional overhead is incurred on the execution time of the application. Second, complete traces of long-running applications are extremely large so that access time to secondary storage becomes the main bottleneck during the analysis. Online trace compression can reduce this overhead but it cannot eliminate it. Overall, software instrumentation for data traces has been shown to increase execution by anywhere between five orders to two orders of magnitude at best [20, 23].

Hardware performance monitoring provides new opportunities to gather performance metrics. For example, obtaining the information from hardware performance counters is extremely low cost and supplies interesting aggregate metrics, including metrics on the performance of the memory hierarchy. However, the aggregate nature of performance counters limits its applicability to only coarse-grained analysis. Finer-grain data is required to pin-point performance bottlenecks in the program, *i.e.*, data traces are needed not just to detect the existence of cache coherence bottlenecks but identify their source and cause.

Hardware-based support for obtaining data traces is beginning to be available on a few high-performance architectures (*e.g.*, on the Itanium-2 and Power architectures). The most sophisticated and flexible, yet readily accessible support at the user level is found on the Itanium-2 [14].

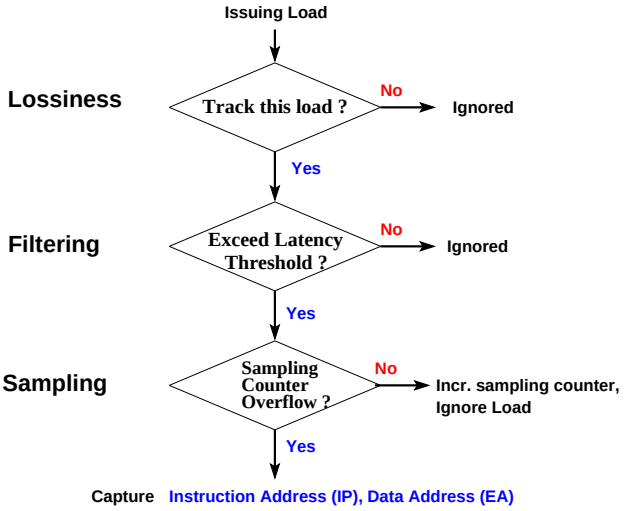
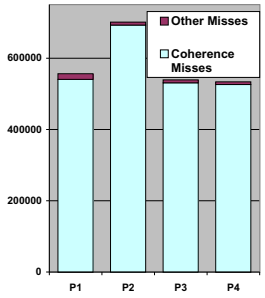


Figure 2. Simplified PMU Operation

3.1 PMU operation

A simplified view of the Itanium-2 Performance Monitoring Unit (PMU) operation for tracing long-latency loads is shown in figure 2. The PMU operation is described in detail in [13]. The PMU supports selective tracking of load instructions based on a latency threshold. However, the PMU cannot track all data cache misses. There are two reasons for this. First, due to hardware restrictions,



(a) Coarse Statistics

No.	Reference	Group	Coherence Misses	Invalidations Received			
				True		False	
				In	Across	In	Across
1	rp[]_Read	**1**	170046	0	0	**156585**	13387
2	rp[]_Read		83509	0	0	**80145**	3529
3	rp[]_Write		43640	0	0	**43305**	3373
4	xp[]_Write		23193	0	0	**22309**	1284
5	num_threads	**2**	44362	**44929**	0	0	0

(b) Per-reference statistics

Figure 1. Characterization for SMG2000

the PMU can only track one load at a time out of potentially many outstanding loads. Second, in order to prevent the same data cache load miss from always being captured in a regular sequence of overlapped cache misses, the PMU uses *randomization* to decide whether or not to track an issuing load instruction. Due to these reasons, the load miss trace available for capture is *lossy*.

If a PMU-tracked load exceeds a user-configured latency threshold value, it qualifies for capture, otherwise it is ignored (*Filtering*). Since the access latencies monotonically increase for cache levels further away from the processor, the latency threshold allows selective capturing of the load miss stream *e.g.*, the L1-D miss stream, the L2 data load miss stream, etc.

Each filtered load increments the PMU overflow counter. By appropriately initializing this counter, the user can vary the *sampling rate* for the captured long-latency load stream (*Sampling*). The Itanium-2 has special support to capture the exact instruction address (IP) and the corresponding data address being loaded (EA) for the sampled long-latency load. In contrast, counter-overflow based sampling on other processor architectures can give misleading instruction addresses for the missing load due to superscalar issue, deep pipelining and out-of-order execution [13].

3.2 Lossy tracing

As described above, the PMU can only capture a fraction of the actual load miss stream, due to hardware limitations. In our experiments with a specially-designed microbenchmark, we observed that the PMU was able to sample only approximately 10% of the missing loads at the highest sampling rate. This illustrates that while hardware support for obtaining metrics is useful, it has its limitations in terms of the granularity of data obtained. Even additional hardware support, such as a fixed-size data trace buffer, would require frequent flushes to memory, which perturbs the memory behavior of applications just as interrupts do.

Lossy tracing, *i.e.*, tracing of only a small subset of data references, such as supported by the Itanium-2, has significantly lower overhead than software-based instrumentation. But can we still obtain sufficiently reliable information about cache coherence bottlenecks based on lossy tracing? Furthermore, the Itanium-2 only supports tracking of loads — but not of store instructions. How can we leverage this limited PMU capability to track stores (essential for modeling coherence traffic) efficiently and without reintroducing prohibitively high runtime overhead? In the following, we detail our approach to hybrid hardware/software tracing that addresses these questions.

3.3 Tracking Stores

The native support of data tracing on the Itanium-2 allows us to capture long latency loads (cache load misses). However, it does not provide native hardware mechanisms to track stores, which are essential for determining cache coherence misses. Hence, we use a hybrid hardware/software approach based on the existing load tracking facility to obtain store traces. We statically rewrite the sequence of instructions to substitute a store with a sequence that, besides performing the store operation, invalidates the cache line of the referenced data before loading it again. Thus, the load results in a cache miss, which can be natively traced by the hardware. We annotate rewritten stores to distinguish them from original load misses, *i.e.*, we can identify them by the IP of the rewritten instruction.

We sketch our store rewrite mechanism here. The store is rewritten into an `xchg` (atomic exchange) instruction, which swaps a register value with the memory location indicated by the address register. Effectively, the exchange results in a memory load and a memory store. Both operations are *forced* to memory and, thus, result in long latencies. The intrinsic long-latency load can now be tracked by the PMU.[1]

This store tracking mechanism incurs only minimal execution overhead, as our results demonstrate. Future hardware may include native support for store tracking, which would a) facilitate our overall efforts and b) alleviate the need for statical binary rewriting.

3.4 Sampling Loads

We track long-latency loads through the Itanium-2 PMU interrupt mechanism. However, a high rate of interrupts results in considerable overhead in execution time. Thus, we investigate several sampling rates, denoted as $OV - r$ for a sampling rate of every r-th event (high-latency tracked load). The Itanium-2 PMU hardware actually facilitates statistical sampling in another way. The PMU *randomizes* whether or not to track a particular load instruction as it is dispatched into the pipeline. This reduces the likelihood that consecutive tracking candidates originate from the same load (IP) and, thus, spreads the tracked loads over multiple references (IPs) in tight loops.

Recall that we observed a 90% loss of data references at even the highest sampling rate (OV-1). Even lower rates (OV-2 and higher) accentuate this loss but, at the same time, considerably reduce the interrupt overhead, as will be shown. Such trace data loss may impact the validity of observed coherence traffic. By skipping references in a trace, a coherence miss may not be observed at all. At other times, the coherence miss may be seen but its correlation to an invalidation may be inaccurate, *i.e.*, the closest store (on another processor) may not be part of the trace such that a much earlier store

[1] The Itanium-2 ISA necessitates several subtleties due to constraints on register types (exchange does not allow fbating-point registers) and short stores (smaller than 64 bits), whose short exchange counterparts clear the most significant bits in a register, even though their value may still be live. We utilize a combination of scratch registers and register spills onto stack where necessary to preserve the original data. The details are beyond the scope of the paper.

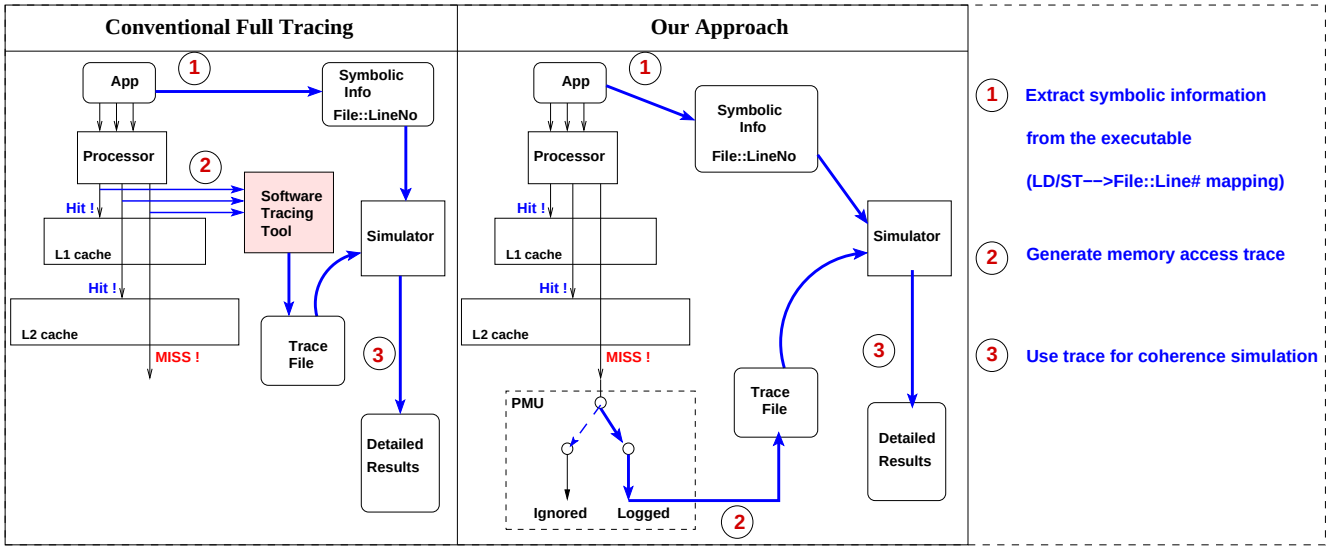


Figure 3. Comparison of Trace-Based Methods

Table 1. Description of Benchmarks

Name	Suite	Data Set	Description
BT	NAS-2.3	Class S, 20/60 iterations	Block triangular solver
CG	NAS-2.3	Class S	Conjugate gradient
EP	NAS-2.3	Class S	Gaussian Random deviates generator
FT	NAS-2.3	32x32x32 grid, 8 iterations	3-D FFT PDE
LU	NAS-2.3	Class S	LU solver
MG	NAS-2.3	32x32x32 grid, 4/20 iterations	Multigrid solver
SP	NAS-2.3	Class S	Pentadiagonal solver
IS	NAS-2.3	Class W	Integer sort
SMG2000	ASCI Purple	10x10x10 grid	Semicoarsening multigrid solver
SPPM	ASCI Purple	35x35x35 grid, 3/10 iterations	Simplified Piecewise Parabolic Method

is falsely implicated. Our experiments assess the validity of coherence analysis under different degrees of "lossiness". They further assess the overhead under varying sampling rates. The experiments study the trade-off between these two parameters

4. Experimental Framework

In the following, we evaluate our framework against a traditional software memory tracing approach.

Figure 3 shows a comparative high-level view of the traditional software-based full-tracing methods *vs.* our lossy-tracing method. In both methods, a memory access trace is generated for the target benchmark. The trace is used offline for incremental coherence simulation. The coherence simulator associates coherence metrics with high-level source code constructs using symbolic information extracted from the target executable. The chief difference between our method and a conventional memory access tracer is in the generation of the memory access trace. The software tracer (left side of Figure 3) logs all memory accesses irrespective of hits or misses. In our hardware-assisted method, accesses which hit in cache are ignored by the PMU. Only a lossy trace of the missing accesses is recorded. Since a vast majority of the accesses are hits, the lossy trace is vastly reduced compared to the original full trace.

For the full-tracing approach, we use the PIN tool on the Itanium-2 for software tracing of memory accesses [26, 18]. The instrumentation points are placed at memory accesses. As the benchmark runs, its memory access trace is captured and written to stable storage. This approach is functionally similar to our previous work [19, 20] using DynInst [5]. In addition, we instrument OpenMP constructs in the benchmark source codes. This allows the coherence simulator to partition the memory access traces and correctly model ordering semantics in the OpenMP program.

The coherence simulator uses the extracted address traces for coherence simulation. Both the PIN-based full traces and the PMU-assisted lossy traces use the same simulator framework. The simulator models the cache hierarchy of the target platform. For this paper, we model the MESI coherence protocol that our target platform uses [13]. Note that when using lossy traces, the trace does not contain the vast majority of accesses which hit in cache and were filtered out by the PMU. Thus, the simulation is not accurate with respect to *absolute* values of uni-processor related metrics (hits, misses, etc.). However, we are interested in the *relative* ranking of source code references, compared to their rankings when using the original trace. The programmer uses hardware counters to first determine that a coherence bottleneck exists, then the lossy framework can be used to obtain the top-ranked references for coherence metrics. The purpose of this work is to assess how close lossy ranking resembles full trace results.

The simulator generates coherence metrics per *reference*, *i.e.*, a source code location (*filename::line_number*). For our evaluation, we consider the following two *coherence metrics*:

- *Invalidations Caused*: The number of times a write from this location caused an invalidation in the cache hierarchy of some other processor.
- *Coherence Misses* encountered: A coherence miss is said to occur when a processor accesses a shared data element whose

cache line state is `Invalid`, indicating that the memory line containing the data element was previously invalidated by some other processor.

These metrics help the programmer to understand the sharing and movement of data among processors. The results generated by the simulator using the full address traces (*full-tracing*) are compared with results generated with lossy address traces obtained with our tracing mechanism (*lossy tracing*).

Our experiments use a set of 10 OpenMP benchmarks for our experiments. The benchmarks are described in Table 1.

The NAS benchmarks are C language OpenMP versions of the original NAS-2.3 serial benchmarks [3] provided by the Omni Compiler group [2]. `SMG2000` and `sPPM` are part of the ASCI Purple benchmark set [1]. The benchmarks are used with comparatively small data sets (class S for NAS), since the software tracing method has prohibitively high run time and trace size overhead with full-sized data sets (while our hybrid approach handles larger sizes without difficulty). `BT`, `IS` and `SPPM` are run with smaller data sets for the software tracing runs, as compared to the lossy tracing runs (see "Data Set" column depicting *data_set_for_tracing_runs / data_set_for_lossy_runs*). Also, for BT and LU, the original code had some manually unrolled loop iterations. We undid this source-level unrolling to decrease the number of source code references taking part in coherence activity (however, the compiler is free to unroll these loops during compilation). For sPPM, we use a larger simulated cache size to allow the benchmark to exhibit coherence activity.

For all benchmarks, the OpenMP scheduling policy for loops was set to `static` scheduling, and the `nowait` clause was removed from OpenMP work-sharing constructs. For lossy tracing, we bound each thread to a distinct processor. The experiments are carried out on a 2-processor Itanium-2 SMP Linux system. All benchmarks were compiled at `-O2` optimization level.

4.1 Design of the Comparison Metric

We evaluate the accuracy and usefulness of the simulator results that use lossy traces. Results in the next section show that lossy traces usually contain far fewer memory accesses, compared to the full trace (reductions of over an order of magnitude). Consequently, lossy tracing could cause the simulator to generate misleading coherence traffic since many of the original accesses are absent. In the following, we describe our quality measures to gauge the accuracy of lossy trace results compared to results obtained using the full trace for the two coherence metrics of *invalidations caused* and *load coherence misses*. We consider only load misses when looking at coherence misses since store misses usually do not stall the issuing processor and, therefore, are not a bottleneck.

We consider two measures for quality:

- **Coverage Fraction**: Results using the lossy trace will give a set of top references with respect to the coherence metric (*e.g.*, for load coherence misses). The coverage fraction indicates what fraction of the total coherence misses these reference account for in the *original* results. A higher coverage fraction indicates that we are capturing most of the application coherence behavior, even when using lossy traces.
- **Number of False Positives**: Due to the lossy nature of the PMU-generated trace, the coherence simulation may attribute coherence traffic to references that do not actually participate in coherence. We count the number of references in the selected lossy-trace based results that have a zero coherence value in original set of results. This measure indicates how potentially *misleading* the lossy-trace based results are.

We generate the above two measures as follows. Each benchmark is run twice. In the first run, we use software instrumentation to extract the full memory access trace from the benchmark execution and use it for coherence simulation. These simulation results constitute the *original* result set for comparison of quality. Then, the benchmark is run again, and lossy traces are obtained using our PMU-assisted method. These traces are similarly used for coherence simulation, and the simulation results generated constitute the *lossy* result set.

The coverage fraction is calculated as follows. Both the original and lossy result sets are sorted in descending order for the metric being considered (load coherence misses or invalidations caused). We select the top-10 references from both set of results. Then,

V1 = Cumulative coverage in the original result set, of the top-10 references obtained using full traces for simulation.

V2 = Cumulative coverage in the original result set, of the top-10 references obtained using lossy traces for simulation.

$Coverage\ Fraction = \frac{V2}{V1} * 100\%$

The coverage fraction compares the coverage obtained with references generated by lossy-trace based simulation *versus* the optimal coverage that is possible with the top-10 references for the coherence metric under consideration. We note that the top-10 references in the lossy trace results may not be identical to the top-10 references selected by the full-trace results. This happens when coherence activity is diffused over many source code references, which end up having very similar coherence metric values.

Also, the number of *false positives* gives an indication of how potentially misleading the lossy-trace based results potentially are. References from the top-10 lossy-trace result set that have a zero metric value in the original results are classified as false positives. A low number of false positives assures that lossy-trace results still correctly represent the actual coherence traffic.

5. Experimental Results

We first report aggregate results from hardware performance counters and discuss the merits as well as limitations of these metrics. We then present our analysis of non-aggregate, reference-based lossy sampling and assess its benefits.

5.1 Hardware Performance Counters

Before using the simulation tool to generate detailed source code correlated statistics, the programmer should determine that a potential coherence bottleneck exists with the benchmark running on the target execution platform. In this section, we describe this characterization using hardware performance counters on our chosen platform (Itanium-2). To our knowledge, this is the first reported use of these counters to characterize shared memory OpenMP coherence traffic.

Performance Events:

The Itanium-2 has 4 performance counters which can be used simultaneously. We monitor the following 4 performance events (from the Itanium-2 manual [13]):

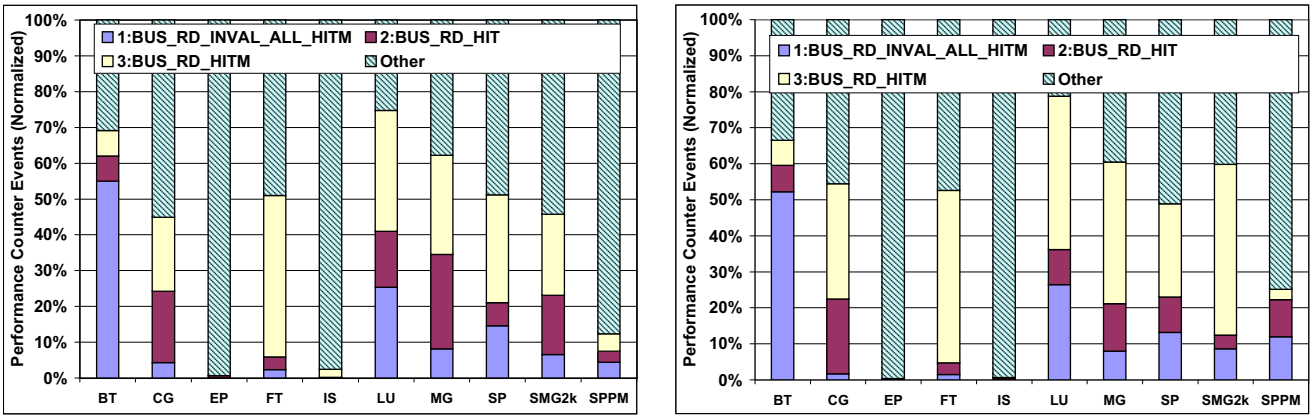
Event 1, BUS_INVAL_ALL_HITM: BUS BRIL (Read-invalidate) and BIL (invalidate) Transaction Results in HITM

Event 2, BUS_RD_HIT: Bus Read Hit Clean Non-local Cache Transactions

Event 3, BUS_RD_HITM: Bus Read Hit Modified Non-local Cache Transactions

Event 4, BUS_MEM_READ_ALL_SELF: Full Cache Line D/I Memory Read, BRIL (Read-invalidate) and BIL (invalidate) transactions

Event 1 counts a processor's *write* cache misses for which the data was found in some other processor's cache whose cache line was in the "Modified" (*i.e.*, dirty) state.



(a) Processor-1

(b) Processor-2

Figure 4. Characterization using Hardware Performance Counters

Events 2 and 3 count the processor's *read* cache misses for which the data was found in some other processor's cache. (HITM stands for "Hit cache line in Modified(M) state").

Each of the above transactions implies bus traffic to transfer the cache line from the remote cache to the requesting processor's caches. The sum of events 1-3 gives an upper bound on the *coherence misses* encountered by the processor.

We compare this coherence miss value to the *total* number of transactions of this type issued by the processor (event 4). A potential bottleneck and optimization opportunity exists if the coherence misses are a significant portion of the total number of coherence transactions.

Characterization:

Each OpenMP thread was bound to a distinct processor. Figures 4(a) and 4(b) show the normalized values for events discussed above, for each processor. The graphs show that many of the benchmarks have significant coherence activity. Event 3 constitutes the largest percentage of transactions in most benchmarks with significant coherence activity. Modified data lines are "pulled" from the local processor cache to the remote processor issuing reads to the same data line. For BT, the bulk of the transactions are due to event 1. This indicates that multiple processors are writing to the same shared data line, causing data to circulate among the local and remote caches. The results are not symmetric across processors for many benchmarks; CG, SMG2000 and SP have distinctly different compositions and magnitudes of coherence misses on processor-2 as compared to processor-1.

Hardware counters can detect significant coherence traffic, as demonstrated above. However, counter values do not indicate the *cause* of the coherence bottleneck. Our lossy-trace-based framework provides detailed source code-correlated statistics that allow the programmer to "drill down" into the bulk statistics and gain insights into the sharing patterns at application source-code level.

In the following, we do not further analyze the EP and IS benchmarks. For EP, the total coherence misses amounts to only 0.6% of the total transactions. This is expected as EP is an "embarrassingly parallel" benchmark and there is little communication between processors. Similarly, IS has very few coherence misses (2.4% of total transactions), we do not analyze it further.

5.2 Evaluating Lossy Tracing

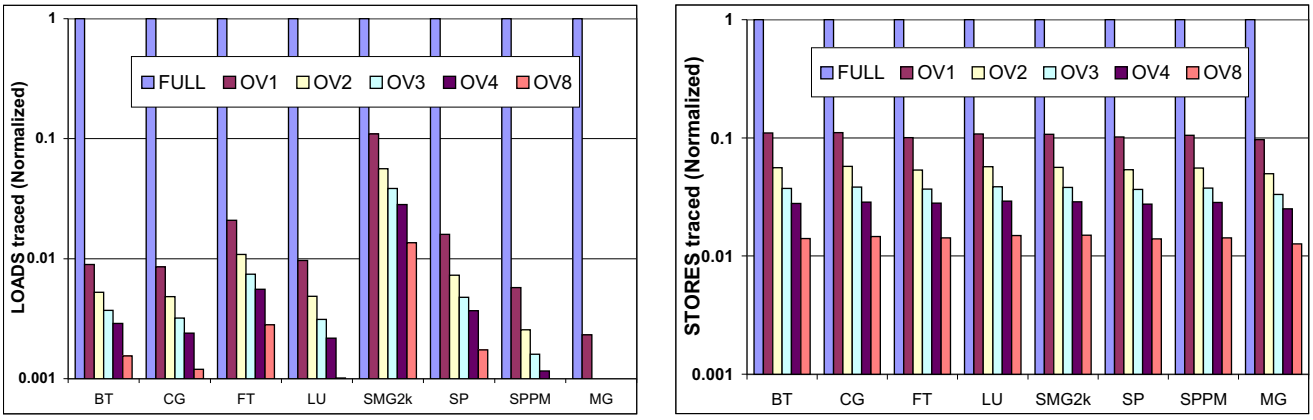
In this section, we evaluate our lossy tracing method with respect to accuracy and overhead. The lossy traces were obtained with the hardware PMU configured at sampling rates of 1,2,3,4 and 8 (OV1 to OV8 in the graphs). We used a cycle threshold of 8 cycles, *i.e*, a load (or xchg) can only qualify for PMU tracking if it takes eight or more clock cycles to complete. This setting corresponds to the access latency of an L2 cache miss for loads on the Itanium-2 [13]. The latency thresholds can only be set in powers of 2, and a threshold less than 8 cycles (*i.e.*, 1, 2 or 4 cycles) is not useful as it would also capture loads which hit in the L1 or L2 caches. These loads do not suffer coherence misses (they are hits), so it is not useful to capture them. Ignoring the load hits also drastically reduces the trace collection overhead and trace sizes, since applications typically have a high cache hit rate.

5.3 Trace Sizes

Figures 5(a) and 5(b) compare the volume of loads and stores traced for the full (software) tracing *vs.* our lossy mechanism at different sampling rates. The y-axis is on a logarithmic scale. Access volumes are normalized to the number of accesses in the full trace.

The graphs show that our method decreases the number of accesses collected by one to two orders of magnitude compared to full tracing. This considerable decrease results from the PMU's ability to discriminate and track only long-latency loads, ignoring the far more frequent low-latency accesses that hit in the L1 and L2 caches. In addition, the PMU has randomization logic which decides whether or not to track a potential long latency load, which further decreases the trace volume. However, this makes the collected trace lossy, *i.e.*, we cannot capture all the instances of loads or xchg instructions which had latencies greater than the cycle threshold (8 cycles).

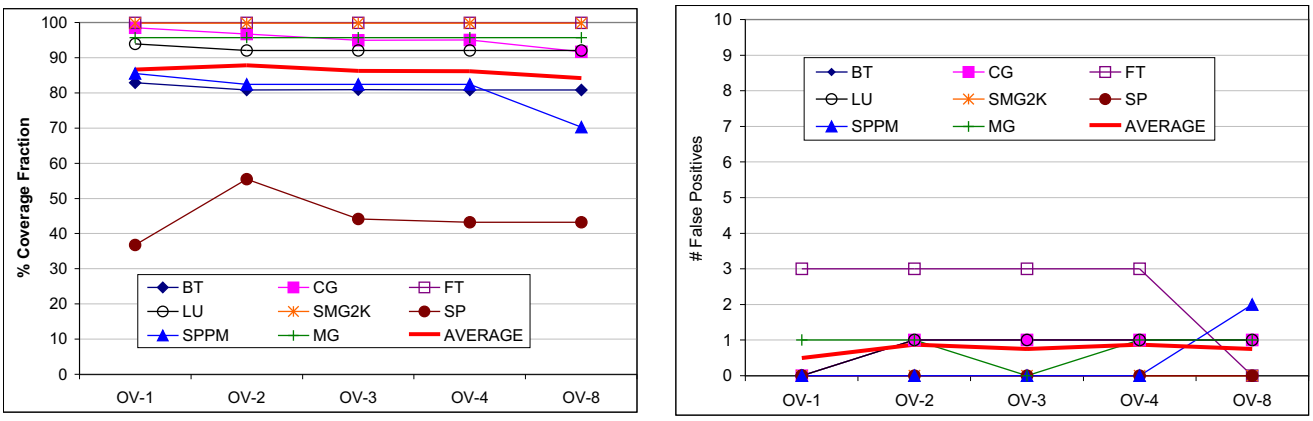
The number of accesses logged linearly decreases for larger sampling intervals. The normalized fraction of stores traced is remarkably similar across benchmarks while there is more variation in the fraction of loads traced. Our annotation mechanism for stores causes this effect: *all* dynamic instances of the annotated stores will miss in cache and will be eligible for PMU tracking. However, only those dynamic instances of loads that miss in cache (*i.e*, long-latency loads) are eligible for PMU tracking; hence, the fraction of loads tracked varies with data cache hit rates of different benchmarks.



(a) Loads Traced

(b) Stores Traced

Figure 5. Memory Accesses Traced, Normalized to # Accesses in Full Trace



(a) Coverage Fraction for Lossy-Tracing vs. Full-Tracing

(b) # False Positives with Lossy-Tracing Approach

Figure 6. Top-10 References Causing Invalidations on Processor 1, PMU Sampling Rates of 1-8

5.4 Accuracy of Results

Figures 6 and 7 depict the accuracy of the results using lossy traces for the two metrics of *invalidations caused* and *coherence misses*. Due to space constraints, only the results for processor 1 are shown. The results for the other processor are similar.

We compare the quality of the results using the yardsticks of *coverage fraction* and *number of false positives*, as described in section 4.1.

A high value for the coverage fraction would indicate that even when using lossy traces, we are able to substantially capture most of the application coherence behavior. A low number of false positives ensures that stand-alone lossy-trace based results are not misleading.

5.4.1 Metric: Invalidations Caused

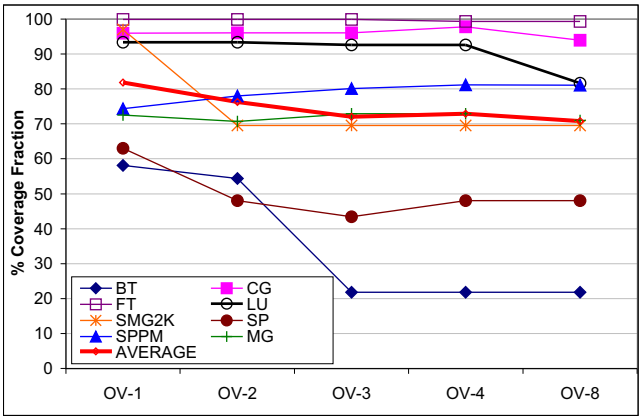
Consider the results for *invalidations* depicted for coverage fraction and the number of false positives in Figures 6(a) and 6(b), respectively. The results are shown for different sampling intervals (OV1 to OV8). For OV1, the coverage fraction ranges from 36-100%, averaging 86%. Except for SP, all benchmarks show a very high coverage fraction of greater than 82%. Looking at the number of false positives, no benchmarks, except for FT and SP, have any false positives at sampling interval OV1. Thus, in most cases, we achieve very high coverage fraction values without false positives in the lossy-trace results.

For SP, the benchmark has a large number of invalidation-causing store references. There are more than 100 source code store references with with a non-zero count of invalidations in the full-trace results. The lossy-trace results are similarly diffused over many store references. The top-10 references selected by lossy-trace results does not include some of the top references from the full trace results, due to which the coverage fraction is low.

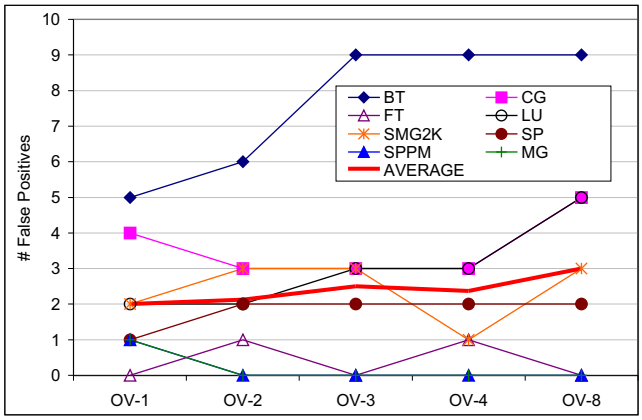
For FT, there are 3 false positives. All these false positives are stores which immediately follow the correct invalidation-causing store. For example:

```
808: xout[k][j][i+ii].real = ...;
809: xout[k][j][i+ii].imag = ...;
```

The first store on line 808 causes the actual invalidations. However, due to lossy-tracing, the first store is sometimes not recorded, but the second store on line 809 is. In this case, the invalidation is mis-attributed to line 809 since both the stores access the same cache line. With advanced dependence analysis, it may be possible to eliminate this type of false positives.



(a) Coverage Fraction for Lossy-Tracing vs. Full-Tracing



(b) # False Positives with Lossy-Tracing Approach

Figure 7. Top-10 References Resulting in Coherence Misses on Processor 1, PMU Sampling rates of 1-8

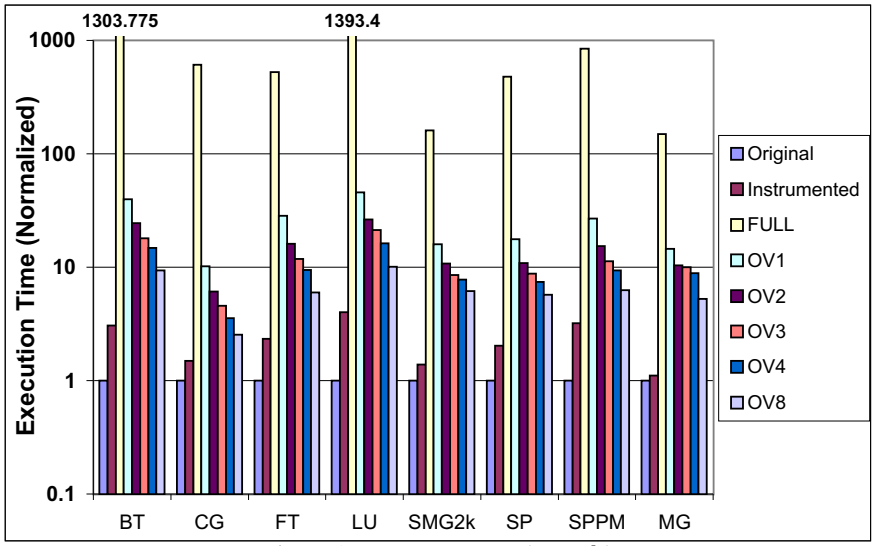


Figure 8. Execution Time (seconds)

Interestingly, as the sampling interval increases, both the coverage fraction and the degree of false positives do not change significantly. Thus, even with smaller traces and less execution overhead (and larger sampling intervals), the quality of the results do not seem to suffer perceptibly.

5.4.2 Metric: Coherence Misses

The results for *coherence misses* are shown in Figures 7(a) and 7(b). The figures show that result quality is dependent on the benchmark. The coverage fraction at OV1 ranges from 57% to 99% with an average value of 81%. SP and BT have comparatively low coverage fraction values of 63% and 58%, respectively. 4 out of the 8 benchmarks (CG, FT, LU, SMG2K) have coverage fraction values greater than 95%.

At OV1, most benchmarks have a low number of false positives (Figure 7(b)), except for BT(5) and CG(4) with an average of 2. Thus, on average, 8 out of the top-10 references generated using lossy-traces are correct. As the sampling interval increases from OV1 to OV8, the average coverage fraction decreases from 81% to 71%, mainly due to a large drop in the coverage fraction value of BT. Similarly, increasing the sampling interval from OV1 to OV8 increases the average number of false positives from 2 to 3, mainly due to a steep rise in the number of false positives for BT (9). The anomalous behavior of BT is explored in more detail below. Except for BT, most other benchmarks have large coverage fraction values and relatively low number of false positives.

5.5 BT

As seen in the last section, lossy-trace based simulation generates very poor coherence miss results for BT. As Figures 7(a) and 7(b) show, BT has a very large number of false positives, even at the highest sampling interval of OV1. As the sampling interval increases, the number of false positives increases, which also causes the coverage fraction value for BT to decrease sharply (since most of the lossy-trace generated references have zero metric value in the full-trace results).

There are multiple causes for BT's poor behavior. First, the simulation results with full traces show that over 90% of the overall coherence misses are *store* misses. However, for our experiment, we only considered the *load* coherence misses since store misses

usually do not stall the issuing processor. The bus cycle breakdown for BT obtained using hardware counters is shown in Figure 4. This confirms that, only for the BT benchmark, store misses are the dominant factor (event BUS_RD_INVAL_ALL_HITM dominates other bus transactions). Due to this, the overall number of load coherence misses is low, and the actual coherence related references get lost in the false positive "noise" references in the simulation results generated with lossy traces.

Second, BT is an array-intensive program. Many of the false positives occur with the following situation:

```
lhs[i][j][k][BB][temp1][temp2]= .....; //Store
......
...... = lhs[i][j][k][BB][temp1][temp2] ; //Load
```

The second load cannot miss in cache since the cache line is brought into the cache (if not already present) by the preceding store. With lossy tracing, it sometimes happens that the first store reference is not traced, but the second load reference is. Due to this, the coherence miss is falsely attributed to the load reference. However, with full traces, the second load reference always hit in cache and, therefore, has zero coherence miss value. Thus, the second load reference is a false positive.

It should be noted that with ideal tracing of the load miss stream, the second load *cannot* be traced since it is a hit. However, the Itanium PMU traces *long-latency* loads, which constitute a superset of the load miss stream (other conditions can cause long-latency loads including TLB misses, bank conflict and queue full conditions). In addition, the tracing framework can perturb the data cache, causing the load reference to miss in cache. Due to a combination of these two factors, we do see the second load reference in the lossy trace, which shows that false positives may occur due to these uncontrolled effects.

5.6 Execution Overhead

The primary motivation behind using PMU-derived lossy traces is the large reduction in runtime execution overheads and generated trace sizes. This reduction results from focusing on long-latency loads — the bulk of the memory accesses hit in the L1 and L2 data cache and are ignored. This allows our method to *scale* and handle much larger, more realistic data sets and benchmark sizes than is possible with past work using full memory access tracing.

Figure 8 quantifies the payoff in terms of reduction of application runtime overhead. It shows the execution time incurred by the benchmarks at different sampling intervals (OV1-OV8) and with full access tracing (FULL) using the dynamic instrumentation tool. The numbers are normalized to the execution time of the original unmodified program. The y-axis is on a logarithmic scale. The "Instrumented" bars show the normalized execution time of the application annotated with our store-annotation scheme described in Section 3 without the use of hardware monitoring.

The improvements in runtime for our lossy tracing method compared to full software-based tracing are very large: from one to over two orders of magnitude. The store instrumentation scheme by itself adds comparatively low overhead. The overhead shows a linear decrease from OV1 to OV8 allowing a trade-off between runtime overhead and the accuracy of results using the lossy trace.

6. Related Work

Several software and hardware-based approaches for shared memory characterization have been described in literature. Gibson *et al.* provide a good overview of the trade-offs of each approach [11].

Several frameworks simulate hardware and architecture state at the instruction level, which inflicts considerable simulation overhead [12, 27]. Our simulator is more lightweight. We only focus on memory hierarchy and coherence simulation. More importantly, these simulators provide only bulk statistics intended for evaluating architecture mechanisms. Our framework is intended to provide *application programmers* with detailed source-level information about the coherence behavior of their programs, enabling program transformations to avoid coherence bottlenecks.

Execution-driven approaches are popular for simulating memory accesses. They utilize annotations of memory access points, which trigger calls to the memory access simulator ([25, 4, 9]. MemSpy [21] and CProf [16] are cache profilers that aim at detecting uniprocessor memory bottlenecks. Lebeck and Wood also applied binary editing to substitute instructions that reference data in memory with function calls to simulate caches on-the-fly [17]. SIGMA uses post-link binary instrumentation and online trace compression [10]. Like us, SIGMA supports tagging of metrics to source code constructs, however, it only supports uniprocessor workloads. As explained in this paper, these approaches have high runtime overhead and do not *scale* to large data sets or long-running real-world programs. In contrast, we use a hardware-assisted approach to collect much smaller traces with considerable savings in run-time overhead.

Several tools provide aggregate metrics obtained at low cost from hardware performance counters. HPCToolkit uses statistical sampling of performance counter data and allows information to be correlated to the program source [22]. A number of commercial tools (Intel's VTune, SGI's Speedshop, Sun's Workshop) also use statistical sampling with source correlation, albeit at a coarser level that HPCToolkit or our approach. Hardware counters complement our lossy-trace based approach: A programmer first uses hardware counters to determine if a coherence bottleneck exists. Then, our framework helps to efficiently extract the lossy trace and to generate detailed source-correlated coherence statistics.

There are many interesting approaches to tuning applications using information provided by hardware counters. Tikir *et al.* describe a profile-driven online page migration scheme using hardware performance counters [24]. Buck *et al.* use the Itanium-2 data tracing PMU support to associate load misses to source code lines and data structures in uniprocessor programs [7]. Buck *et al.* also compare different hardware mechanism for detecting uniprocessor memory hierarchy bottlenecks [6]. Satoh *et al.* study dataflow techniques to analyze data sharing patterns at compile time for OpenMP programs [28]. While these approaches focus on application tuning, our contribution is on efficient large-scale performance analysis. Thiffault *et al.* compare the cost of dynamic and static software instrumentation for large-scale OpenMP and MPI programs [30]. We, in contrast, promote hardware-assisted sampling due to overheads resulting from software instrumentation in general.

7. Conclusion

This paper details a novel hardware-assisted approach to determine coherence bottlenecks. Our mechanism uses the Itanium-2 hardware performance monitor PMU that accurately associates data addresses with load instructions and filters interrupts for these instructions based on a latency threshold. In addition to filtering for latency, the PMU also provides sampling frequency support. We combine the PMU support with an efficient software technique to capture store data addresses to provide a lossy-trace mechanism. We apply this mechanism to a large set of OpenMP benchmarks in order to explore the trade-off between accuracy and overhead, in terms of trace size and runtime slowdown.

Our lossy-trace mechanism provides a low runtime overhead method to identify coherence bottlenecks in OpenMP applications. The lossy traces have two possible sources of inaccuracy: coherence misses omitted due to sampling and the omission of a store that actually causes a coherence miss. Our results demonstrate that

lossy tracing can reduce runtime overhead by two orders of magnitude compared to full-trace software-based approaches while retaining a high degree of accuracy for most benchmarks.

Acknowledgments

The comments of the anonymous referees helped improve the quality of the paper.

References

[1] Asci purple codes. http://www.llnl.gov/asci/purple, 2002.

[2] C versions of nas-2.3 serial programs. http://phase.hpcc.jp/Omni/benchmarks/NPB, 2003.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[4] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. Proteus: A high-performance parallel-architecture simulator. In *Proceedings of the SIGMETRICS and PERFORMANCE '92 International Conference on Measurement and Modeling of Computer Systems*, pages 247–248, New York, NY, USA, June 1992. ACM Press.

[5] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[6] B. R. Buck and J. K. Hollingsworth. Using hardware performance monitors to isolate memory bottlenecks. In ACM, editor, *Supercomputing*, pages 64–65, 2000.

[7] B. R. Buck and J. K. Hollingsworth. Data centric cache measurement on the intel itanium 2 processor. In ACM, editor, *Supercomputing*, 2004.

[8] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, University of Wisconsin, Madison, July 1996.

[9] H. Davis, S. R. Goldschmidt, and J. Hennessy. Multiprocessor simulation and tracing using tango. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume II, Software, pages II–99–II–107, Boca Raton, FL, Aug. 1991. CRC Press.

[10] L. DeRose, K. Ekanadham, J. K. Hollingsworth, , and S. Sbaraglia. SIGMA: A simulator infrastructure to guide memory analysis. In *Supercomputing*, Nov. 2002.

[11] J. Gibson. *Memory Profiling on Shared Memory Multiprocessors*. PhD thesis, Stanford University, July 2003.

[12] C. Hughes, V. Pai, P. Ranganathan, and S. Adve. Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors. *IEEE Computer*, 35(2):40–49, February 2002.

[13] Intel. *Intel Itanium2 Processor Reference Manual for Software Development and Optimization*, volume 1. Intel, 2004.

[14] Intel Corp. *Intel Itanium2 Processor – Reference Manual*, May 2004.

[15] A. Krishnamurthy and K. Yelick. Optimizing parallel programs with explicit synchronization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–204, 1995.

[16] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *Computer*, 27(10):15–26, Oct. 1994.

[17] A. R. Lebeck and D. A. Wood. Active memory: A new abstraction for memory system simulation. *ACM Transactions on Modeling and Computer Simulation*, 7(1):42–77, Jan. 1997.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

[19] J. Marathe, F. Mueller, T. Mohan, B. R. de Supinski, S. A. McKee, and A. Yoo. Metric: Tracking down inefficiencies in the memory hierarchy via binary rewriting. In *International Symposium on Code Generation and Optimization*, pages 289–300, Mar. 2003.

[20] J. Marathe, A. Nagarajan, and F. Mueller. Detailed cache coherence characterization for openmp benchmarks. In *International Conference on Supercomputing*, pages 287–297, June 2004.

[21] M. Martonosi, A. Gupta, and T. Anderson. Memspy: analyzing memory system bottlenecks in programs. In *Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 1–12, 1992.

[22] J. Mellor-Crummey, R. Fowler, and D. Whalley. Tools for application-oriented performance tuning. In *International Conference on Supercomputing*, pages 154–165, June 2001.

[23] T. Mohan, B. R. de Supinski, S. A. McKee, F. Mueller, A. Yoo, and M. Schulz. Identifying and exploiting spatial regularity in data memory references. In *Supercomputing*, Nov. 2003.

[24] J. K. H. Mustafa M. Tikir. Using hardware counters to automatically improve memory performance. In ACM, editor, *Supercomputing*, 2004.

[25] A.-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The augmint multiprocessor simulation toolkit: Implementation, experimentation and tracing facilities. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 486–491, Washington - Brussels - Tokyo, Oct. 1996. IEEE Computer Society.

[26] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *37th International Symposium on Microarchitecture*, Dec. 2004.

[27] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE parallel and distributed technology: systems and applications*, 3(4):34–43, Winter 1995.

[28] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP compiler for an SMP cluster. In *EWOMP '99*, pages 32–39, Sept. 1999.

[29] S. Satoh, K. Kusano, and M. Sato. Compiler optimization techniques for openMP programs. *Scientific Programming*, 9(2-3):131–142, 2001.

[30] C. Thiffault, M. Voss, S. T. Healey, and S. W. Kim. Dynamic instrumentation of large-scale mpi/openmp applications. In *International Parallel and Distributed Processing Symposium*, Apr. 2003.