Analysis of Cache Coherence Bottlenecks with Hybrid Hardware/Software Techniques

JAYDEEP MARATHE and FRANK MUELLER, North Carolina State University
BRONIS R. de SUPINSKI, Lawrence Livermore National Laboratory

Application performance on high-performance shared-memory systems is often limited by sharing patterns resulting in cache-coherence bottlenecks. Current approaches to identify coherence bottlenecks incur considerable run-time overhead and do not scale.

We present two novel hardware-assisted coherence-analysis techniques that reduce trace sizes by two orders of magnitude over full traces. First, hardware performance monitoring is combined with capturing stores in software to provide a lossy-trace mechanism, which is an order of magnitude faster than software-instrumentation-based full-tracing and retains accuracy. Second, selected long-latency loads are instrumented via binary rewriting, which provides even higher accuracy and control over tracing but requires additional overhead.

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

Additional Key Words and Phrases: Hardware performance monitoring, dynamic binary rewriting, program instrumentation, cache analysis, SMPs, coherence protocols

1. INTRODUCTION

Recent high performance computing platforms incorporate multiple processors connected by a fast interconnection system. Many of these systems provide different degrees of shared memory abstraction. Examples of this approach include clusters of SMPs with multiple processing chips sharing memory over a bus-based coherence protocol (*e.g.*, Intel Xeons), chip-multiprocessors (*e.g.*, the IBM Power5) and large-scale cache coherent NUMA machines (*e.g.*, the SGI Altix).

Scientific codes on such machines incorporate data parallelism, *i.e.*, multiple threads of the program work on different parts of the data set in parallel. The underlying *coherence protocol* in hardware ensures that each processor always accesses the most recent version of the data element. Application performance and scalability is affected to a significant degree by the sharing pattern of data among the application threads and its impact on the cache coherence system.

Authors' address: J. Marathe and F. Mueller, Department of Computer Science, North Carolina State University, Raleigh, NC 27695-7534, e-mail: mueller@cs.ncsu.edu, phone: +1.919.515.7889
B.R. de Supinski, Lawrence Livermore National Laboratory, Center for Applied Scientifi c Computing, L-557, Livermore, CA 94551
Please see Section Acknowledgements for prior conference publication.

Sharing patterns that result in frequent invalidations followed by subsequent coherence misses represent cache coherence bottlenecks with significant performance penalties. However, the complexity of the hardware makes it difficult for programmers to assess the effects on shared resources, specifically those imposed by cache coherence traffic between processors, for the multitude of architecture variations (bus-based SMPs vs. CMPs vs. directory-based SMPs). Thus, users need a scalable performance analysis methodology to detect coherence bottlenecks.

Coherence behavior can contribute significantly to wall-clock time. Mixed-mode scientific parallel applications often support a hybrid MPI+OpenMP model, but tend to be more optimized for MPI performance, and only to a lesser extent for their OpenMP usage. Our previous work addresses this problem and pin-points potential coherence bottlenecks [Marathe et al. 2004]. We showed that code transformations can result in up to 73% improvement in wall-clock time for large-scale NNSA ASC benchmarks[pur 2002], closely resembling production code. In addition, with the advent of multi-core architectures, there is growing interest in using OpenMP for shared-memory parallelism. In this paper, we describe efficient techniques to understand sharing behavior of such multi-threaded programs.

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 [Burger et al. 1996; Hughes et al. 2002; Nguyen et al. 1996; Brewer et al. 1992; Davis et al. 1991] to the operating system interface [Rosenblum et al. 1995]. Past work on the performance tuning concentrates on program analysis to derive optimized code [Krishnamurthy and Yelick 1995; Satoh et al. 2001].

More recent work on identifying coherence bottlenecks is based on tracing memory accesses *via* dynamic binary rewriting [Marathe et al. 2004; Tao and Weidendorfer 2004]. This approach can reduce the trace collection overhead by an order of a magnitude or more over conventional hardware simulators. But the execution time overhead is still significant compared to the un-instrumented performance of the application. Due to this, the approach does not easily scale with larger data sets. It is useful for hot-spot analysis over short periods of time, but it is infeasible for the analysis of the entire execution of long-running applications. In practice, this may discourage programmers from using such an analysis tool. These past approaches are slow because of the reliance on purely software-based techniques to obtain data traces, either by means of slow hardware simulations or *via* software instrumentation with significant overhead per access point.

In this paper, we present novel low-cost hardware-assisted methods to determine coherence bottlenecks in shared-memory applications. These methods use existing processor features to reduce collected trace sizes and execution overheads by a significant degree.

Our first method, PMU-based tracing, uses the Itanium-2 hardware performance monitor (PMU), which accurately associates data addresses with load instructions and filters interrupts for these instructions based on a latency threshold. 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.

Our second method, targeted tracing, provides more control of the tracing process. In this approach, we first use the PMU latency-based filtering to cut down on the number of

instructions to instrument. This reduced set of instructions is instrumented using binary rewriting. Each load instance is timed, and only loads that exceed a software-defined latency threshold are captured. Stores are sampled with different sampling intervals.

We evaluate both methods with a large set of OpenMP benchmarks. We compare them against a naïve software instrumentation-based approach that captures the entire access trace of the program. We explore the tradeoffs between accuracy of the results based on reduced traces obtained with our methods *vs.* the size of the collected trace and the overhead of trace collection. A method is considered accurate if it generates results that closely resemble the results generated using the full trace. Section 4.1 details the metrics, namely coverage fraction (most frequent coherence misses we detect *vs.* those in the full trace) and number of false positives (references that we false identify as coherence misses).

We show that both of our methods reduce the number of loads captured by more than two orders of magnitude over the full trace. The PMU-based method is more than an order of a magnitude faster than software-instrumentation based full-tracing. Its accuracy, a metric first introduced below and detailed in Section 4.1, is high accuracy on most benchmarks. Targeted tracing provides even higher accuracy and control over the tracing process, at the cost of additional overhead compared to the PMU-based method.

To the best of our knowledge, our methods significantly outperform any prior approaches. They make cache coherence analysis feasible for long-running applications for the first time.

In this paper, we make the following contributions:

—Two new hardware-assisted tracing methods for coherence analysis;

—PMU-based Method: Design of a PMU-based method to filter out irrelevant accesses and to reduce trace collection cost by an order of magnitude;

—Targeted Method: Design of software-based tracing enhanced by PMU support to *prune* the set of instrumented access points to reduce the tracing cost;

—Use of cycle accurate hardware timing registers to filter out accesses unrelated to coherence to cut trace sizes by two orders of magnitude (for Targeted Method);

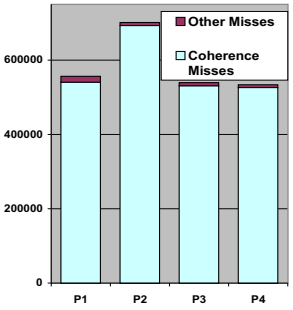
—Definition of two coherence trace accuracy metrics: coverage fraction (most frequent coherence misses detected *vs.* those in the full trace) and number of false positives (references that misidentified as coherence misses);

—Comparison of our two novel tracing methods to software-instrumentation based full-tracing including evaluations of their execution overheads, of their trace sizes and of their accuracy;

—Demonstration that PMU-based tracing usually has high accuracy and reduces the trace size and collection cost by orders of magnitude;

—Demonstration that targeted tracing also decreases the trace sizes by two orders of magnitude and significantly reduces execution overhead while generating even more accurate results PMU-based tracing, but at a relatively higher execution cost.

The paper is structured as follows. First, we demonstrate the usefulness of detailed source code-correlated coherence metrics. We then describe the two hardware-assisted trace capture approaches. Next, we sketch the experimental setup and results. Finally, we contrast our approach with prior work and summarize our contributions.



(a) Coarse Statistics

No.	Reference	Coherence Misses	Invalidations Received			
			True		False	
			In	Across	In	Across
1	rp[]_Read	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	44362	**44929**	0	0	0

(b) Per-reference statistics

Fig. 1. Characterization for SMG2000

2. SOURCE-CORRELATED STATISTICS

In prior work, we used a full-tracing approach to extract complete access traces from OpenMP applications [Marathe et al. 2004]. These traces were 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, we motivate the need for source-code correlated coherence characteristics.

Consider SMG2000, a production-quality OpenMP benchmark from the ASCI Purple suite [pur 2002]. The example stems from our prior work [Marathe et al. 2004]. 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, we cannot answer these questions only with aggregate metrics; we must "drill-down" and associate coherence metrics with 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 incurred many coherence misses, and received many false-sharing invalidations.

3. EXTRACTING MEMORY ACCESS TRACES

We evaluate a variety of access tracing schemes with different degrees of hardware assistance. We start with a purely software-instrumentation based scheme. Then, we describe a pure hardware scheme that leverages the PMU's capabilities to filter out irrelevant accesses and generates results based on a fraction of the remaining accesses. Finally, we describe a composite *targeted tracing* method that uses hardware profiling and timing mechanisms to

focus the software memory capture only on interesting memory accesses.

Pure software-based instrumentation: Software instrumentation can be inserted either by the compiler, a static binary rewriter or via dynamic binary rewriting. The instrumentation intercepts memory access instructions and captures the resulting memory access trace.

PMU-based lossy tracing: We introduce a new lossy tracing mechanism that uses the Itanium-2 performance monitoring unit (PMU) capabilities to capture long-latency loads. The latency threshold can be increased to make the capture mechanism more selective (*i.e.*, only capture the L1 miss stream or L2 miss stream).

Hardware-assisted targeted software tracing: Naïve software instrumentation can generate a large volume of accesses, which is difficult to store and to process. We introduce a composite method that first uses the PMU to filter out load instruction addresses that never miss in cache. In a later run, we only instrument the remaining load access points. The software tracer times each tracked access and only captures accesses exceeding a software-defined latency threshold.

We evaluate these methods with respect to three properties: *cost*, *trace size* and *accuracy*. Their definitions are given below.

Cost: This measures the execution time overhead inflicted on the target program by the trace capture mechanism. Software-instrumentation based tracing has been shown to increase execution time by anywhere between five to two orders of magnitude at best [Marathe et al. 2004; Mohan et al. 2003].

Trace Size: This measures the number of memory accesses in the trace. Each access is described by two fields: the address of the instruction that generated the memory access, and the data address that the instruction was accessing. As discussed before, software-instrumentation based tracing leads to very large trace sizes so that access to secondary storage becomes the main bottleneck during the analysis. Online trace compression mechanisms can reduce this overhead, but they cannot eliminate it.

Accuracy: This measures the degree of closeness between the results generated using a tracing method *vs.* the results generated using the full memory access trace. Section 4.1 details the metrics, namely coverage fraction and number of false positives. For example, when considering coherence misses, the coverage fraction measures the number of coherence misses recognized by using a tracing method versus the number of coherence misses recognized using the full trace (for a selected set of top source code locations). The false positives are the source code locations that do not appear in the full trace-based results, but do appear in the lossy trace based results. False-positives are *misleading*, and the number of false positives should be low for a lossy trace-based method to be effective.

We now describe the PMU-based hardware lossy tracing scheme and the hardware-assisted targeted tracing scheme in detail.

3.1 Method I: PMU-based Lossy Tracing

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 to identify their source and cause.

Hardware-based support for obtaining data traces is beginning to be available on a few

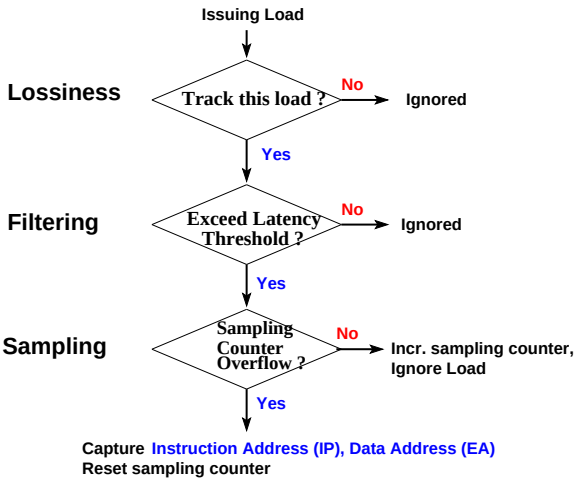


Fig. 2. Simplified PMU Operation

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 [Intel Corp. 2004].

3.1.1 *PMU Operation.* A simplified view of the Itanium-2 Performance Monitoring Unit (PMU) operation for tracing long-latency loads is shown in Figure 2; full details are available elsewhere [Intel 2004]. The PMU supports selective tracking of load instructions based on a latency threshold.

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

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 [Intel 2004].

3.1.2 *Lossy Tracing.* All loads that miss in cache must take more than a fixed number of cycles to execute (the cache miss latency). Ideally, by setting the cycle threshold of the PMU below this fixed value, we could capture the cache miss stream. However, in our experiments with a specially designed microbenchmark, we observed that the PMU was able to capture only 10% of the total loads that missed in cache, even at the highest sampling rate. There are two reasons for this. First, the PMU can only track one load at a time out of potentially many outstanding loads due to hardware restrictions. Second, the PMU uses *randomization* to decide whether or not to track an issuing load instruction in order to prevent the same data cache load miss from always being captured in a regular sequence of overlapped cache misses. Thus, the load miss trace available for capture is *lossy*.

Furthermore, the Itanium-2 only supports tracking of loads — but not of store instructions. Can we obtain sufficiently reliable information about cache coherence bottlenecks given the lossy nature of the trace? How can we leverage the 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.

3.1.3 *Tracking Stores.* 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. Since the `xchg` involves a load operation, the PMU can track this instruction. On our platform, we observed that the `xchg` always took more cycles to execute than the published latency cycles for a L2 cache hit. This allows us to filter out most of the L2 load hits and still capture a lossy fraction of the instrumented stores (*i.e.*, xchg instructions). [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 static binary rewriting.

There are several other ways in which stores could potentially be captured. Our second tracing method (Section 3.2) uses software tracing to capture a fraction of stores. This method has higher overhead than the mechanism discussed above (since it is implemented fully in software), but allows finer control over trading off overhead *vs.* the volume of stores captured. We explore this aspect further in Section 7. Stores could also be captured by exploiting virtual memory protection mechanisms. In this method, a range of virtual memory pages could be protected against write access. A store to data in any of these pages will cause an access violation, and the store access can be captured in the fault handler. However, the virtual memory page would then need to be unprotected, the write re-executed and the page protected again — all within the handler routine. Performance results of such a method are beyond the scope of this paper but the overhead of calling a handler, issuing two more system calls and communicating the value to write to the handler (since it is typically not available inside the handler), possibly requiring instruction disassembly, seems prohibitively high.

3.1.4 *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

[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 signifi cant 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 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 is falsely implicated. Our experiments assess the validity of coherence analysis under different degrees of "lossiness". By increasing the sampling interval, we can decrease the overhead of tracing — at the cost of having fewer trace records available for simulation. This may impact the quality of the trace-based coherence simulation. Our experiments explore the tradeoff between these two factors (overhead *vs.* impact of the increased lossiness on result accuracy).

3.2 Method II: Hardware-assisted Targeted Software Tracing

In our experiments, we will show that the PMU-based tracing scheme is very fast, compared to software tracing. However, the *lossy* nature of the generated scheme may lead to a decrease in accuracy. Our second method *emulates* the action of the PMU in software, by timing individual loads, and only capturing the loads that exceed the cycle threshold. Thus we expect to filter out the bulk of load accesses that hit in cache, but without the lossiness that comes with the PMU-based approach.

This process works as follows. In the first step, we *reduce* the potential set of load access points that must be instrumented. We run the target program without software instrumentation and set the PMU latency threshold greater than the latency (in cycles) for an L3 cache hit (64 cycles in this paper). We find the set of load instructions that do not appear in the PMU-logged trace at all. We remove these load instructions from further analysis since they will not contribute to coherence traffic. The rationale for this decision is as follows. Consider the set of load instructions that do not occur at all in the PMU-generated trace. There are two possible causes. First, these loads could occur frequently but mostly hit in the L3 cache (or higher levels of cache). These loads can be ignored as they do not cause coherence traffic since they hit in processor-local caches. Second, some of these load instructions have high L3 miss rates but occur infrequently enough that our lossy PMU-based trace does not have a single occurrence of them. Since these loads execute infrequently, we can ignore them without affecting the accuracy of the resulting coherence simulation. Recall that the accuracy metric only considers the top references resulting in coherence misses.

In the second step, we use a dynamic binary rewriter to instrument the remaining load access instructions and re-run the program. For every instrumented load, the instrumentation uses a high-resolution timer to measure the number of processor cycles needed to load from the memory location accessed by the load.[2]

[2]The instrumentation first reads the high-resolution timer. It then executes a load access to the data location accessed by the instrumented load instruction and immediately uses the loaded register in a dummy instruction. This use causes the in-order Itanium2 processor to stall until the data is loaded from memory. The difference between the two readings was experimentally found to approximate the number of cycles required to load the data from memory closely, even when disregarding the overhead to read the timer.

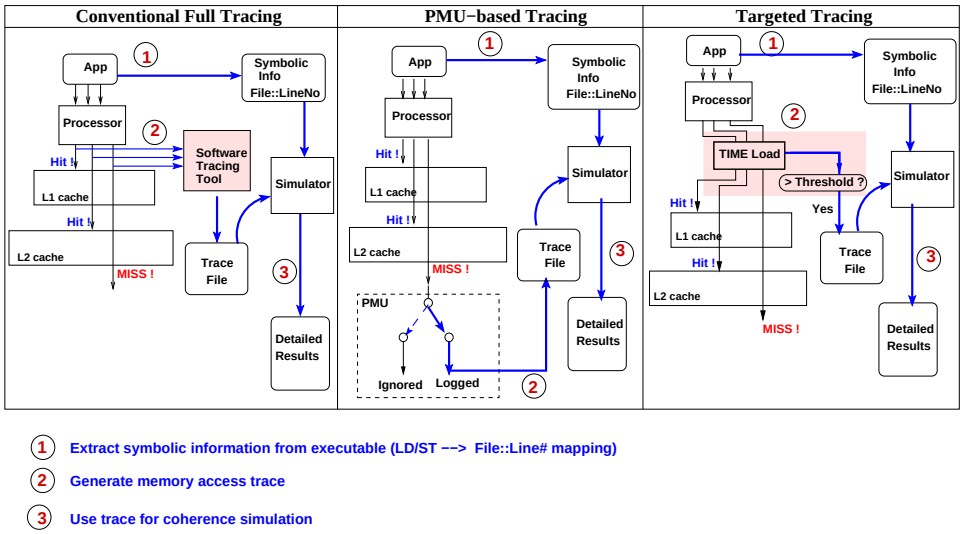


Fig. 3. Comparison of Trace-Based Methods

The access is logged only if it takes more than a software-defined cycle threshold. The cycle threshold is set high enough (64 cycles) so that most of the loads that hit in the processor's caches (and, therefore, do not generate coherence traffic) are filtered out.

We capture a super-set of the cache load miss stream, *i.e.*, there are some loads in the captured trace that would hit in the processor's caches in the original target execution. This occurs due to two reasons. First, in addition to missing in cache, loads can also be delayed due to other factors such as a TLB miss or bus contention. Second, the instrumentation mechanism *perturbs* the processor caches and causes additional misses to occur. As we shall see in the experimental evaluation, even with these caveats, the number of loads captured is still reduced by multiple orders of magnitude over the original full-sized trace.

Using timing thresholds, we can filter out many load accesses that hit in cache. The case of *store* instructions is different. Even stores that hit in cache can cause invalidations to occur in remote processors' caches if the memory line being written to is shared. Since the cache line states are not visible to software, there is no way to know whether a particular stores caused an invalidation or not. Thus, we potentially must capture all the stores that occur; the later coherence simulation will indicate whether the store actually resulted in an invalidation. In our experiments, we vary the software sampling rates for the capture of stores and evaluate its impact on the accuracy of the coherence simulation results.

4. EXPERIMENTAL FRAMEWORK

In the following, "reduced-trace" refers to hardware-assisted methods (PMU-based tracing and targeted tracing), which filter out some accesses from the trace. "Full-trace" refers to the naïve software-instrumentation based tracing, which captures the entire memory access trace of the target program.

Figure 3 shows a high-level comparison of the full-trace based method and the two hardware-assisted methods that we introduce in this paper. In all of our methods, a memory access trace is generated for the target benchmark. The trace is used offline for incre-

mental 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 the methods lies in the generation of the memory access trace. The naïve software tracer (extreme left in Figure 3) logs all memory accesses irrespective of hits or misses. PMU-based and targeted tracing filter out some access, as discussed before.

For full-tracing and targeted tracing, we use the PIN tool on the Itanium-2 for software tracing of memory accesses [Patil et al. 2004; Luk et al. 2005]. The instrumentation points are placed at memory accesses. As the benchmark executes, its memory access trace is captured and written to stable storage. This approach is functionally similar to our previous work [Marathe et al. 2003; Marathe et al. 2004] using DynInst [Buck and Hollingsworth 2000a]. In addition, we instrument OpenMP constructs in the benchmark source codes for all the methods. This instrumentation 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. The simulator models the cache hierarchy of the target platform. For this paper, we model the MESI coherence protocol that our target platform uses [Intel 2004]. Note that when using reduced traces (targeted tracing and PMU-based tracing), the trace does not contain the vast majority of accesses that hit in cache and were filtered out by the PMU. In addition, the PMU-generated trace is lossy. So, the simulation with PMU traces is not accurate with respect to cache *capacity* constraints — since we do not have all references in the trace, we cannot model the cache replacement policy accurately. For instance, it is possible that a memory line that would have been flushed from the cache in reality is still resident in the cache when another processor writes to it. In this case, we would inaccurately count this event as an invalidation (since data was found in the cache). However, as our results show, even with this constraint our results are quite accurate for the benchmarks we study.

Due to the above reasons, the simulation with reduced traces may not be 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 reduced trace methods can be used to obtain the top-ranked references for coherence metrics. The purpose of this work is to assess how close these results are to the full trace results.

The simulator generates coherence metrics per *reference*, *i.e.,* a source code location (*filename::line_number*). Our evaluation considers these two *coherence metrics*:

—*Invalidations Caused*: The number of times a write generated by this source code location caused an invalidation in the cache hierarchy of some other processor. A write causes an invalidation if the data line is also cached in another processor in shared state. The remote reference is invalidated while the local one becomes exclusive.

—*Coherence Misses* encountered: A coherence miss occurs 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 full-trace based results are compared to the results obtained using reduced traces generated by the hardware-assisted frameworks.

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

Table I. Description of Benchmarks

Name	Suite	Data Set	Description
BT	NAS-2.3	Class S, 20/60 iter.	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 iter.	3-D FFT PDE
LU	NAS-2.3	Class S	LU solver
MG	NAS-2.3	32x32x32 grid, 4/20 iter.	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 iter.	Simplified Piecewise Parabolic Method

The NAS benchmarks are C language OpenMP versions of the original NAS-2.3 serial benchmarks [Bailey et al. 1991] provided by the Omni Compiler group [nas 2003]. `SMG2000` and `sPPM` are part of the ASCI Purple benchmark set [pur 2002]. The benchmarks are used with comparatively small data sets (class S for NAS) since the full-trace software tracing method used for comparison has prohibitively high run time and trace size overhead with full-sized data sets. `BT`, `IS` and `SPPM` are run with larger data sets for the PMU-based tracing method. (see "Data Set" column depicting *data_set_for_software_tracing_runs / data_set_for_pmu-based_tracing_runs*). Also, the original code for BT and LU 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 can 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 PMU-based 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 reduced traces. Results in the next section show that reduced traces usually contain far fewer memory accesses compared to the full trace (reductions of over an order of magnitude). Consequently, the reduced traces may 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 reduced 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 reduced 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.

—**Number of False Positives**: Due to the large number of accesses missing from the reduced traces, the coherence simulation may incorrectly attribute coherence traffic to some reference. We count the number of references in the selected reduced-trace based results that have a zero coherence value in original set of results.

We perform two different comparisons. First we compare the full-trace results against the lossy trace-based results obtained with PMU-based tracing. Next, we compare the full-trace results against the reduced trace-based results obtained with targeted tracing.

Full trace Results vs PMU-based lossy results: 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. The simulator output for a particular input memory trace consists of a list of *references*. For the experiments in this paper, a *reference* is a source code location, *i.e.*, a unique filename::line_number identifier. Associated with each reference are the values for each metric (load coherence misses or invalidations caused). We have two sets of simulator results — one generated using the full trace (obtained via software instrumentation) and the other generated using the lossy, PMU-generated trace. Both these result sets are sorted in descending order for the particular metric being considered (load coherence misses or invalidations caused). Then, we select the top-10 references from each result set and compare the *coverage*[3] obtained by each.

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.

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

What do the quality metrics signify?: The coverage fraction compares the coverage obtained with references generated by lossy-trace based simulation *vs.* the optimal coverage that is possible with the top-10 references for the coherence metric under consideration. 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.

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.

Full trace Results vs. Targeted tracing reduced trace results: We use the same quality metrics of *coverage fraction* and *false positives* for this comparison. We characterize

[3]To calculate coverage, consider the following example: If the top-10 references together accounted for X load coherence misses out of a total of Y load coherence misses recorded by the simulator, then the coverage value is $\frac{X}{Y}$.

the accuracy of the results for different store sampling intervals. The store sampler uses a random number generator to sample store instances with varying probability. We experiment with store sampling probabilities of 1.0, 0.25, 0.10 and 0.05, which correspond to capturing an average of all, 25%, 10% and 5% of stores in the full trace.

5. 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 four performance counters which can be used simultaneously. We monitor the following four performance events [Intel 2004]:

Event 1, BUS_INVAL_ALL_HITM. BUS BRIL (Read-invalidate) and BIL (invalidate) Hit Modified Non-local Cache Transactions.

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 one 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.

Events two and three 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").

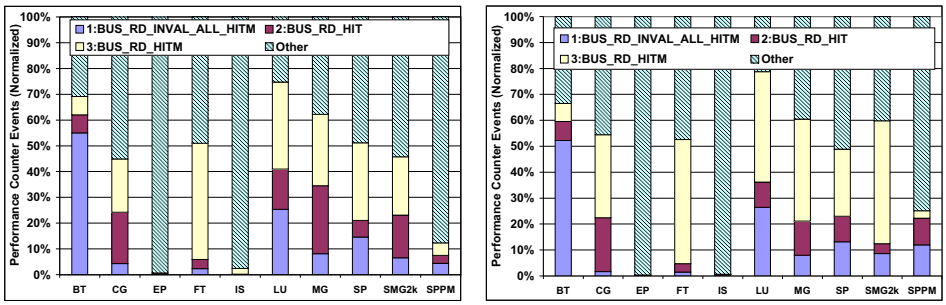
Each of the above transactions implies bus traffic to transfer the cache line from the remote cache to the requesting processor's caches. The sum of events 1-3 gives an upper bound on the *coherence misses* encountered by the processor.

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

Characterization

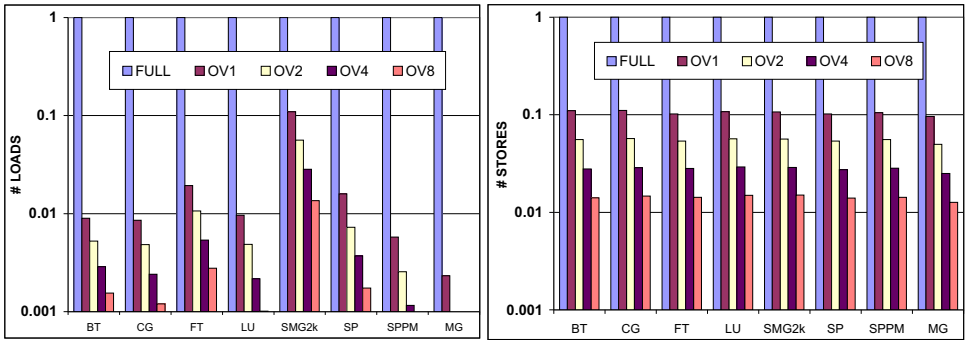
Each OpenMP thread was bound to a distinct processor. Figures 4(a) and 4(b) show the normalized values for the four events for each processor. The graphs show that many of the benchmarks have significant coherence activity. Event 3 constitutes the largest percentage of transactions in most benchmarks 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. However, counter values do not indicate the *cause* of the coherence bottleneck. Our lossy-trace-based framework



(a) Processor-1 (b) Processor-2

Fig. 4. Characterization using Hardware Performance Counters



(a) Loads Traced (b) Stores Traced

Fig. 5. Memory Accesses Traced with PMU-based tracing, Normalized to Number of Accesses in Full Trace

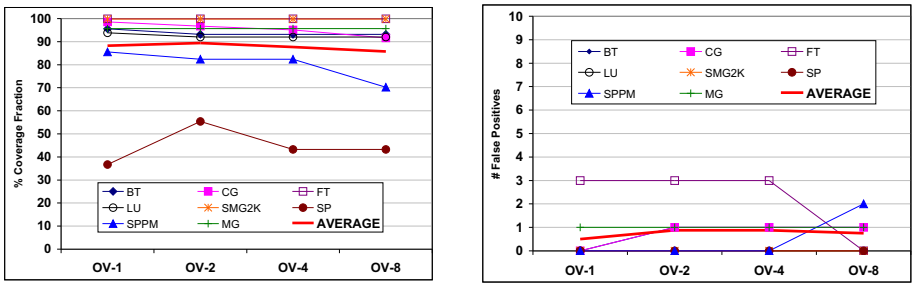
provides detailed source code-correlated statistics that provide a “drill down” into the bulk statistics and, thus, insights into the sharing patterns at application source-code level.

For EP, the total coherence misses amount 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). Thus, we do not further analyze the EP and IS benchmarks.

6. EVALUATING PMU-BASED LOSSY TRACING

In this section, we shall evaluate the PMU-based Lossy Tracing method with respect to cost and accuracy. *Cost* comprises the execution overhead of tracing and the volume of accesses that are captured. *Accuracy* measures how good the generated lossy trace is compared to using the full trace.

We obtained lossy traces with the hardware PMU configured at sampling intervals 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



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

(b) Number of False Positives with PMU-based Lossy-Tracing Approach

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

(or xchg) can only qualify for PMU tracking if it takes eight or more cycles to complete, which corresponds to the access latency of an L2 cache miss for loads on the Itanium-2 [Intel 2004]. 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. [4]

6.1 Trace Sizes

Figures 5(a) and 5(b) compare the volume of loads and stores traced for full tracing *vs.* PMU-based tracing at different sampling intervals. 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. 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. The actual sampling rate of stores will depend on several factors including: the mix of floating point and integer instructions; the temporal rate of memory accesses; and the ratio of reads to writes. However, some filtering will always occur.

[4] We observed that the bulk of the "xchg" accesses, which represent store instrumentation, take only 12 cycles to execute on our test platform (probably because the instrumentation induces a bank conflict that delays the xchg). Setting the latency threshold higher than 8 cycles would cause most of the instrumented stores to be filtered out. In future work, we intend to explore alternate store instrumentation schemes that do not have this limitation.

6.2 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 accuracy of the results using the yardsticks of *coverage fraction* and *number of false positives*, as described in section 4.1.

6.2.1 *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 benchmark, except for FT and SP, has 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 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 do 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 that immediately follow the correct invalidation-causing store. For example:

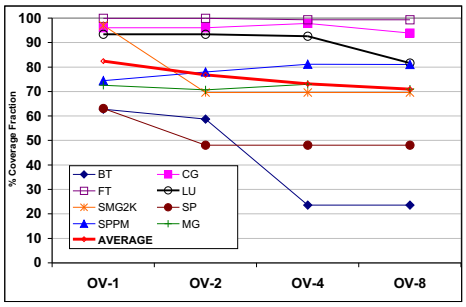
```
808: xout[k][j][i+ii].real = ...;
809: xout[k][j][i+ii].imag = ...;
```

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. Advanced dependence analysis might help eliminate this type of false positives.

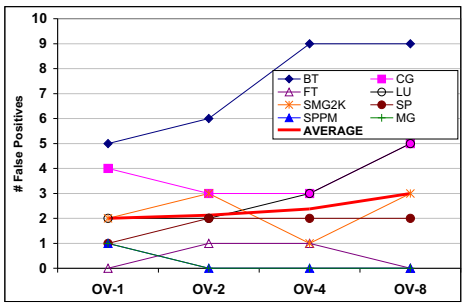
Interestingly, the coverage fraction and the degree of false positives do not change significantly as the sampling interval increases. Thus, accuracy does not degrade perceptibly even with smaller traces and less execution overhead (and larger sampling intervals).

6.2.2 *Metric: Coherence Misses.* Figures 7(a) and 7(b) show that accuracy metrics for *coherence misses*, for which accuracy 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. Four of the eight 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 two. Thus, on average, eight 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 two to three, 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.



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



(b) Number of False Positives with PMU-based Lossy-Tracing Approach

Fig. 7. Top-10 References Resulting in Coherence Misses on Processor 1, PMU Sampling Intervals of 1-8, PMU-based tracing

6.3 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 store misses are the dominant factor for only the BT benchmark (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 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, the load reference can be traced when the store is not. Thus, the coherence miss may be falsely attributed to the load reference. However, with full traces, the load reference always hits in cache and, therefore, has zero coherence miss value. Thus, the load reference is a false positive. [5]

[5]It should be noted that with ideal tracing of the load miss stream, the 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

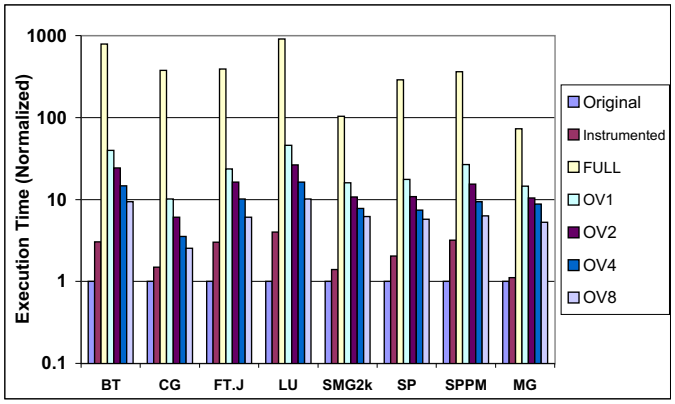


Fig. 8. Execution Time: Full-tracing *vs.* PMU-based tracing

6.4 Execution Overhead

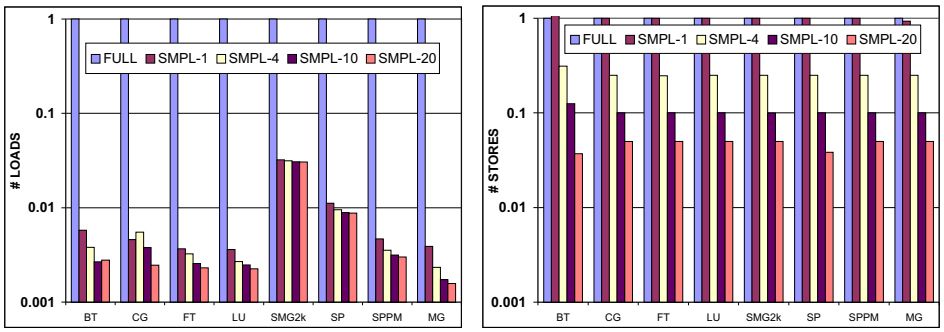
Figure 8 quantifies the payoff in terms of reduction of application runtime overhead. It shows the execution time incurred by the benchmarks at different sampling intervals (OV1-OV8) and with full access tracing (FULL) using the dynamic instrumentation tool. The numbers are normalized to the execution 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.

7. EVALUATING TARGETED TRACING

In the preceding section, we evaluated the PMU-based tracing with respect to execution overhead, trace sizes and accuracy. The PMU-based tracing is very efficient with respect to execution overhead and trace sizes. However, due to the lossy nature of the trace, the number of false positives is large for a few benchmarks (*e.g.*, for BT). In the following, we evaluate an alternate method, *targeted tracing*, that uses software instrumentation to trace memory accesses but uses hardware features to cut down on the accesses traced. We explore the tradeoff between having more control over the accesses that we capture (since that is now decided in software) *vs.* the accuracy of the resulting trace and the execution overhead of capturing the trace.

As described before, we first run the un-instrumented program with a large latency threshold (64 cycles). The load instruction addresses that do not appear in this generated trace do not miss in cache and can be ignored for coherence purposes. In the second pass, we instrument only the reduced set of load instructions (which have appeared in the

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.



(a) Loads Traced

(b) Stores Traced

Fig. 9. Memory Accesses Traced with Targeted tracing, Normalized to Number of Accesses in Full Trace

PMU-generated trace). Further, for each load instance, we measure the time it took for the load to complete. If it is greater than a software-defined threshold (64 cycles), we trace the load, else we ignore it. Thus, we essentially emulate the hardware PMU's capability to filter out loads by *latency*, but without the PMU's lossiness.

For store instructions, we experiment with different software-defined sampling intervals. The C-library `srand()` and `random()` functions are used to capture stores with probabilities of 1, 0.25, 0.1 and 0.05. The default random number generator is very accurate; the corresponding number of stores captured are approximately 100%, 25% , 10% and 5% of the total stores, respectively. The sampling intervals are shown in the graphs as SMPL-1, SMPL-4, SMPL-10 and SMPL-20 (*i.e.*, all stores, 1 in 4, 1 in 10 and 1 in 20 stores on average are captured).

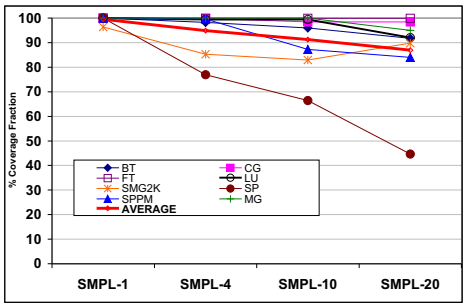
7.1 Trace Sizes

Figures 9(a) and 9(b) compare the number of loads and stores traced with targeted tracing compared full tracing. The values are normalized to the number of accesses in the full trace. The y-axis is on a logarithmic scale. Figure 9(a) show that for all but one benchmark (SMG2K), targeted tracing cuts down on the number of loads in the trace by more than two orders of magnitude over the full trace. The number of loads traced decreases slightly as the software store sampling interval is increased from SMPL-1 to SMPL-20, probably due to the reduced cache perturbation of the instrumentation. The large decrease in loads over the full trace also confirms that the tracing framework does not significantly perturb the data caches, in that many of the original loads still hit in cache.[6]

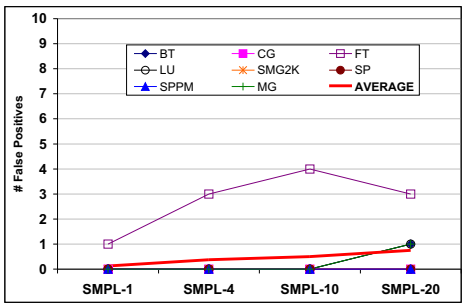
Figure 9(b) shows that the number of stores captured decreases as expected when the software sampling interval is increased from SMPL-1 to SMPL-20. The impact of this lossy tracing of stores is explored in the following paragraphs.

It is instructive to compare these figures *vs.* the corresponding ones (Figures 5(a) and 5(b)) for PMU-based tracing. The number of loads traced for OV-1 in PMU-based tracing are comparable to the number of loads captured by targeted tracing, even with the differ-

[6] A large perturbation of the cache would have been indicated by observing that almost all the target loads miss in cache, thus increasing their latency of access and qualifying for capture.

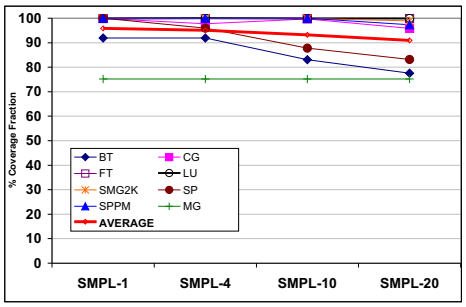


(a) Coverage Fraction for Targeted Tracing vs. Full-Tracing

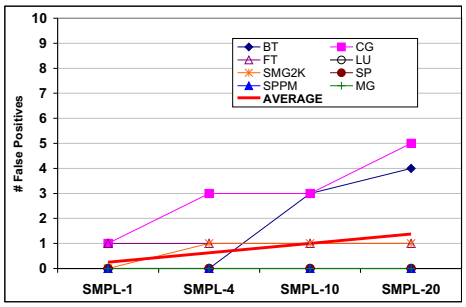


(b) Number of False Positives with Targeted Tracing Approach

Fig. 10. Top-10 References Causing Invalidations on Processor 1, Store Sampling Rates of 1,4,10,20



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



(b) Number of False Positives with Targeted Tracing Approach

Fig. 11. Top-10 References Resulting in Coherence Misses on Processor 1, Store Sampling Rates of 1, 4, 10, 20

ing latency thresholds used (8 cycles for the PMU-based tracing, 64 cycles for targeted tracing). With the lower latency threshold, many more loads qualify for capture by the PMU, but this is offset by the lossiness of the PMU, which captures only a fraction of the eligible loads. Also, observe that the number of stores captured at OV-1 and OV-2 for the PMU-based method is very close to the number of stores captured by the software sampler in targeted tracing at sampling intervals SMPL-10 and SMPL-20, respectively.

7.2 Accuracy of Results

As for the PMU-based results, we evaluate accuracy with the two yardsticks of *coverage fraction* and *number of false positives*. Figures 10 and 11 show these yardsticks applied to the metrics of *invalidations caused* and *coherence misses*, respectively. The results shown are for processor-1.

7.2.1 *Metric: Invalidations Caused.* Figures 10(a) and 10(b) show the coverage fraction and number of false positives for this metric, over different store sampling intervals

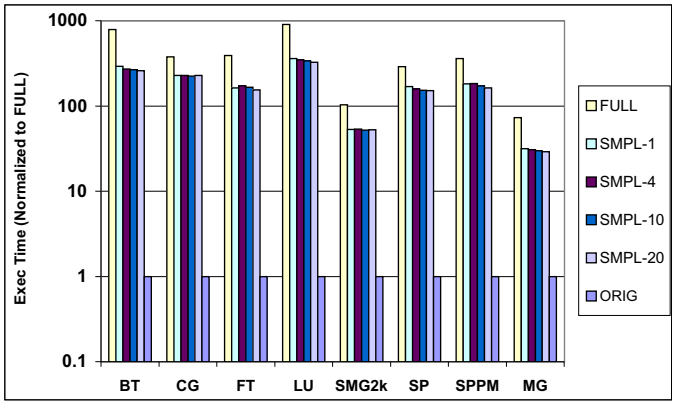


Fig. 12. Execution Time: Full-tracing *vs.* Targeted tracing

ranging from SMPL-1 to SMPL-20. At SMPL-1, when all stores are captured, the coverage fraction is extremely high (average: 99.54%) and the number of false positives is zero for all benchmarks except for FT. As the sampling interval increases to SMPL-20, the average coverage fraction reduces to 86% while the coverage for SP decreases more significantly. This behavior of SP is similar to its performance with PMU-based tracing (Figure 6(a)).

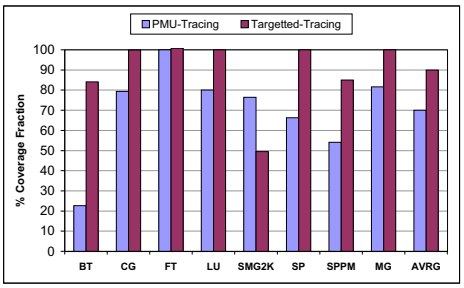
Even at sampling interval SMPL-20, most benchmarks have no false positives, except for FT (3 false positives) and SP (1 false positive). The false positives for FT are the same references as for PMU-based tracing (see Section 6.2.1). These false positives again appear because of lossiness in the store trace that is captured (Section 6.2.1) at sampling intervals other than SMPL-1.

7.2.2 *Metric: Coherence Misses.* Figures 11(a) and 11(b) show the coverage fraction and number of false positives for coherence misses over the different store sampling intervals. The average coverage fraction ranges from 95% at SMPL-1 to 90% at SMPL-20. At SMPL-1, all benchmarks have a coverage fraction greater than 91%, except for MG (75%). The average number of false positives is less than one (Figure 11(b)) at SMPL-1, and increases to 1.3 at SMPL-20.

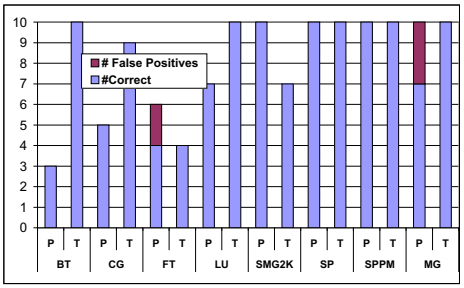
The behavior for BT and CG is interesting. At SMPL-1, *i.e.*, capturing all stores, the number of false positives for these benchmarks is 0 and 1, respectively. With increasing lossiness of stores beyond SMPL-1, the number of false positives increases sharply. Finally, at SMPL-20, CG has 5 false positives and BT has 4. Similarly, with PMU-based tracing, these benchmarks show many false positives even at OV-1 (Figure 6(b)). Thus, these benchmarks are very sensitive to the degree of lossiness of stores.

7.3 Execution Time

Figure 12 compares the execution time for targeted tracing *vs.* the full tracing. The numbers are normalized to the original execution time for each benchmark. The saving in execution time, compared to full tracing, range from 40% to 68%. For each benchmark, the increasing sampling intervals do not much impact the execution time. This is because the trace framework has been optimized so that most of the time per access is spent in deciding whether to capture the access or not (using the random number generator). With



(a) Coverage Fraction



(b) Correct References vs. False Positives (P=PMU-based Tracing, T=Targeted Tracing)

Fig. 13. Coverage and False Positives for PMU-based and Targeted Tracing with Respect to True-sharing Invalidations

a simpler sampling strategy (*e.g.* using a counter) we saw increasing savings in execution time as the store sampling interval increased, but we do not report those results here.

The execution time savings, though useful, are not as dramatic as those for PMU-based tracing. The chief benefit of targeted tracing over full tracing comes in the large trace size reductions (especially loads), which makes it easier and faster to use the trace for offline activities, *e.g.*, for incremental coherence simulation in our case.

8. COMPARING TRUE SHARING AND FALSE SHARING

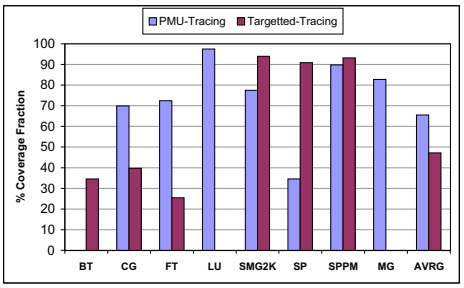
We maintain per-cache-line bit vectors indicating which parts of the cache line have been accessed (either by a load or a store) from the attached processor. This allows us to classify stores that cause invalidations as either *true-sharing invalidations* or *false-sharing invalidations*. When a store reference on a processor causes an invalidation to occur in some other processor's cache and when at least one byte in the range of addresses being written to (determined by the storage width of the store instruction) has been accessed by the other processor, we classify the invalidation as a *true-sharing* invalidation. Otherwise, we classify the invalidation as a *false-sharing* invalidation.

Classifying invalidations into these subtypes gives the programmer additional insight into the sharing behavior of the program. False-sharing invalidations, in particular, indicate potential for optimization by data layout or code transformations[Marathe et al. 2004].

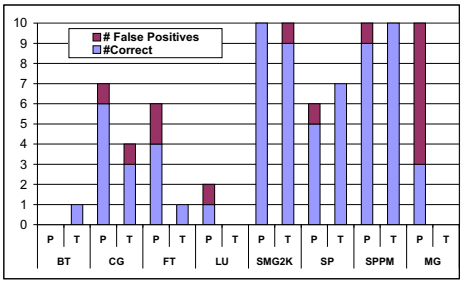
In the results below, the PMU-based runs used a sampling interval of 1 (OV-1). The targeted tracing used store sampling of 1 (SMPL-1), *i.e.*, all stores are captured.

8.1 Comparing True Sharing Invalidations-Caused

We compare the two reduced-trace based methods (PMU-based tracing, targeted tracing) against the results obtained from full tracing for *coverage* and *number of false positives*. For comparing true-sharing, we only select references from the reduced-trace based results that accounted for at least 2% of the overall true-sharing invalidations in the simulation results up to a maximum of 10 references. By setting this lower threshold, we are trying to reduce the number of false positives that are generated, potentially at the expense of reducing the coverage fraction. Reducing the number of false positives is more important to ensure that the stand-alone reduced-trace based results are not misleading.



(a) Coverage Fraction



(b) Correct References vs. False Positives (P=PMU Tracing, T=Targeted Tracing)

Fig. 14. Coverage and False Positives for PMU-based and Targeted Tracing with Respect to False-Sharing Invalidations

Table II. True and False-Sharing Invalidations Caused Measured with Full Traces

Name	TrueSharing	FalseSharing	Total	FalseSharing %
BT	278538	48646	327184	**14.86**
CG	20059	4194	24253	**17.29**
FT	20642	428	21070	**2.03**
LU	95850	1852	97702	**1.89**
SMG2K	90205	286346	376551	**76.04**
SP	176863	385050	561913	**68.52**
SPPM	16521	52970	69491	**76.22**
MG	4585	167	4752	**3.51**

Figures 13(a) and 13(b) compare the coverage fraction and number of false positives. We observe:

—Consider Figure 13(b): in some cases, our strict selection criteria enables less than 10 references to be selected from the reduced trace results (*e.g.*, for BT, only 3 total references are selected with PMU-based tracing). The small number of references usually reduces coverage values for these benchmarks (*e.g.*, BT has only 22% coverage fraction with PMU-based tracing).

—Consider PMU-based tracing: it produces few false positives and only MG and FT have *any* false positives. The coverage fraction is high, (except for BT) averaging 70%. For BT, our thresholding scheme allowed only three references to be selected leading to the low coverage value.

—Consider targeted tracing: it produces *no* false positives. The coverage values are also much higher (averaging 89%) than those for PMU-based tracing. Thus, targeted tracing produces results very similar to full tracing.

8.2 Comparing FalseSharing Invalidations-Caused

The benchmarks we considered had distinctly different false-sharing behavior. Table II shows the total number of true and false sharing invalidations caused by references in

processor-1 based on full address traces. For FT, LU and MG, the false sharing invalidations constitute less than 4% of the total invalidations. In contrast, for SMG2K, SP and SPPM, the false sharing invalidations account for more than 68% of the total invalidations.

Ideally, when using reduced traces, we have two expectations. For benchmarks with a low fraction of false sharing invalidations (FT, LU, MG), it is more important to reduce the number of *false positives* than to obtain a high coverage value. For the benchmarks with large fraction of false-sharing invalidations (SMG2K, SP, SPPM), it is important that we obtain a high coverage value in addition to a low number of false positives.

When using reduced traces for finding false-sharing invalidations, we came across a unique problem. Due to the lossy nature of the trace, many of the actual true-sharing invalidations were classified as *false* sharing invalidations. This occurred because the target cache had no record of access to the cache line by the attached processor (due to trace lossiness). This problem is exacerbated by another factor. Since our reduced trace methods attempt to avoid tracing loads that hit in cache, we lose potential information, *i.e.*, we cannot tell which which parts of the cache line were actually accessed by the processor. Typically, the load access that missed and brought the memory line into the cache is recorded, but the subsequent accesses to the other bytes in the memory line (*self and cross reuse* [Wolf and Lam 1991]) may be lost since they hit in the cache and are filtered out. This limitation only applies to loads as all stores are enabled for tracing in both PMU-based and targeted tracing.

We attempted to overcome this problem by being more strict when selecting references from the reduced trace based results. For a reference to be selected as a false-sharing reference, it must have at least twice the number of false-sharing invalidations as true-sharing invalidations (in order to compensate for some of the "fake" false-sharing invalidations, which are actually true-sharing invalidations). In addition, the reference must account for at least 2% of the total false-sharing invalidations and for at least 2% of the *total* invalidations. This restriction ensures that we only qualify references that occur somewhat frequently, otherwise we cannot rely on their values. As before, our focus is on reducing the number of *false positives*, at the potential expense of lowered coverage fraction.

Figures 14(a) and 14(b) show the coverage fraction and number of false positives caused by false-sharing invalidations.

We observe:

—Consider PMU-based tracing: it produces few false positives for all benchmarks other than MG while its coverage fraction varies. For BT, our restrictive selection policy did not allow even a single reference to be selected so that the coverage value is 0. The coverage value is high for all other benchmarks (except for SP) averaging 65%.

—Consider targeted tracing: it produces almost *no* false positives its coverage values are high for benchmarks with significant false sharing (*e.g*, SPPM) and lower for other benchmarks. For LU and MG, the coverage is 0 since not a single reference was selected from the reduced trace-based results. Since both LU and MG have a very low fraction of false-sharing invalidations, this result is expected (Table II).

—Consider MG: PMU-based tracing produces many false positives. Table II shows that this benchmark has the lowest absolute number of false-sharing invalidations. With PMU-based tracing, many of the references that had only true-sharing invalidations in reality had a large number of false-sharing invalidations. Our reference selection restrictions were ineffective in this case. Thus, PMU-based tracing can give misleading results

Table III. Ratio of False-Sharing Invalidations to Total Invalidations for the Selected References in PMU Tracing and Full Tracing

Name	# Selected Refs	False Sharing % of Selected Refs in: PMU-Tracing	False Sharing % of Selected Refs in: Full Tracing	Similar ?
BT	0	0	0	Yes
CG	7	89.28	11.61	**No**
FT	6	79.03	1.44	**No**
LU	2	16.21	1.81	**No**
SMG2K	10	61.73	48.12	Yes
SP	6	22.69	30.57	Yes
SPPM	10	43.96	44.85	Yes
MG	10	66.94	2.84	**No**

when there inherently exists no false sharing in the benchmark, though that occurred for only one benchmark in our evaluation.

8.3 Limitations of Reduced-Trace Based Simulation

We have demonstrated that the number of false positives generated in the reduced-trace based results is low for false-sharing invalidations with a strict selection criteria. This indicates that, in general, reduced-trace based results *generate the same list of references* as those obtained with full-trace-based simulation. If we want to use reduced-trace based results in a stand-alone fashion, we must also answer another question: For the given list of selected references, does it make sense to optimize these references for false sharing? In other words, *does the false sharing incurred by the selected references play a dominant role in the overall invalidations generated for the program*? It will not be worthwhile optimizing the references to reduce false sharing if they do not dominate.

This question can be answered by comparing the accumulated false-sharing invalidations incurred by the selected references *vs.* the total invalidations recorded for the benchmark. Table III shows such a comparison. First, our selection criteria is used to select the top set of references (up to 10 references) for PMU-based tracing for false-sharing invalidations (column 2). Then, the number of false-sharing invalidations for this set as a percentage of the total invalidations recorded in the simulation is calculated (column 3). A similar calculation is done for the same set of references for full tracing (column 4). The ideal case has similar values in the two columns, *i.e.*, either both are low or both are high. On the other hand, if PMU-based tracing produces a much higher percentage of false sharing invalidations than full tracing then the *reduced-trace based results are exaggerating the degree of false sharing for the benchmark.* As the table shows, the values are similar for four benchmarks (BT, SMG2K, SP, SPPM) and dissimilar for the remaining four (CG, FT, LU, MG). The four benchmarks that have dissimilar values have an inherently low degree of false sharing, as seen from Table II. When these benchmarks are simulated with PMU-based trace, there are many cases where *true-sharing* invalidations are incorrectly classified as *false-sharing* invalidations (see Section 8.2). Thus, if a benchmark has inherently low false sharing, reduced-trace simulation can give misleading results.[7] For benchmarks that do have significant false sharing, the reduced-trace based simulation gave correct results

[7]Targeted tracing produces very similar results to PMU-based tracing when compared to full tracing.

(*e.g.*, SPPM, MG, SMG2K).

Thus, we can only trust the reduced-trace based results for false-sharing invalidations caused if we *know* that the benchmark has significant false-sharing behavior. A straightforward method to do this would be to add an additional performance counter to count the false-sharing invalidations. The idea is to first run the program without tracing overhead and check whether the program *has* significant false sharing. If so, our lossy-trace based framework can be used to find the actual source code references that account for this false sharing quickly and accurately. Adding this hardware support would require keeping track of which parts of the cache line have been accessed by the attached processor. This could be achieved by keeping a bit vector for each cache line. The space overhead for the bit vector can be reduced by "chunking", *i.e.*, making a single bit responsible for multiple bytes in the cache line (trading off precision for space overhead). Let C be the cache line size in bytes. If we assign 1 bit for 4 consecutive bytes (the typical size of an unsigned integer), the space overhead per cache line would be (C/4)/8, *i.e.*, only 3.1%. We plan to explore this approach in future work.

9. COMPARING TARGETED TRACING AND PMU-BASED TRACING

In the earlier sections, we compared the execution cost, trace sizes and accuracy for each of the hardware-assisted methods to full tracing. PMU-based tracing has at least an order of magnitude less execution overhead compared to full tracing. The overhead also decreases linearly with increasing sampling intervals (OV-1 to OV-8). However, PMU-based tracing lossiness causes some false positives for particular benchmarks. The false positives are more pronounced for the coherence misses metric with an average of 2 false positives (out of 10) at OV-1. Some benchmarks are especially sensitive to the lossy nature of the trace and have many false positives (5 false positives for BT and 4 false positives for CG, at OV-1). In addition, when comparing false-sharing invalidations, almost every benchmark had at least one false positive, and some had even more (MG).

With targeted tracing, we have more control over the tracing process at the cost of higher execution overhead (compared to PMU-based tracing). Still, the method saves 40% to 68% execution overhead compared to full tracing. What we lose with execution overhead, we gain with trace size and result accuracy. The reductions in the trace size are comparable to the ones achieved with PMU-based tracing (over two orders of magnitude over full tracing). Targeted tracing has greater accuracy than PMU-based tracing, especially for the coherence miss metric. For this metric, PMU-based tracing has an average coverage value of 81% and an average of 2 false positives at OV-1 compared to 95% average coverage and 0.25 average false positives at SMPL-1 for targeted tracing. Also, targeted tracing was generally superior to PMU-based tracing when comparing true and false sharing. For true sharing, targeted tracing had no false positives at all and had much larger coverage values. For false sharing, the number of false positives was also much lower than for PMU-based tracing.

10. 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 [Gibson 2003].

Several frameworks simulate hardware and architecture state at the instruction level,

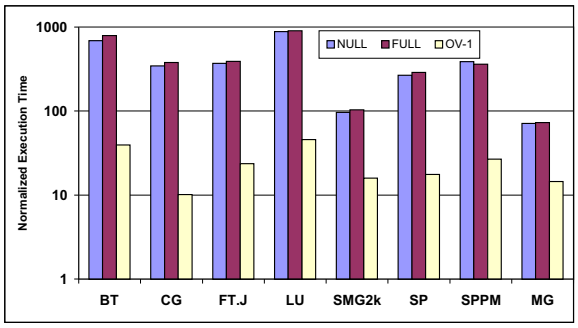


Fig. 15. Execution Overhead Comparison

which incurs considerable simulation overhead [Hughes et al. 2002; Rosenblum et al. 1995]. 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 ([Nguyen et al. 1996; Brewer et al. 1992; Davis et al. 1991]. MemSpy [Martonosi et al. 1992] and CProf [Lebeck and Wood 1994] 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 [Lebeck and Wood 1997]. SIGMA uses post-link binary instrumentation and online trace compression [DeRose et al. 2002]. Like us, SIGMA supports tagging of metrics to source code constructs, however, it only supports uniprocessor workloads. These approaches use *software* instrumentation to capture the trace. We measured the cost of just the trace instrumentation without processing the trace at all using the PIN dynamic instrumentation framework. The `NULL` series in Figure 15 denotes this value. This is the *minimum* overhead that execution-driven simulators must incur, as it measures only the cost of instrumenting the program for tracing. The execution overhead of our hardware-assisted tracing is denoted as `OV-1`, and the cost of the software tracing scheme including the trace storage overhead is shown by `FULL`. Thus our technique has at least one *order of magnitude* less overhead compared to execution driven simulators. Our approach can therefore *scale* to large data sets or long-running real-world programs at significantly less cost. In the closest related work, Tao and Weidendorfer report a multiprocessor cache simulation approach for OpenMP programs [Tao and Weidendorfer 2004] based on the SIMT multiprocessor simulation tool [Tao et al. 2003]. They use binary rewriting to extract the complete memory access trace using Valgrind [Nethercote and Seward 2003] similar to the full-tracing approach we compare against in this paper. The authors report a slowdown factor of 1000 over the original unmodified program. We are not only an order of magnitude faster, but, our trace sizes are over two orders of magnitude smaller, enabling much faster simulation.

Several tools provide aggregate metrics obtained at low cost from hardware performance counters. HPCToolkit uses statistical sampling of performance counter data and allows in-

formation to be correlated to the program source [Mellor-Crummey et al. 2001]. 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 determines if a coherence bottleneck exists through hardware counters. Then, our framework extract the lossy trace efficiently generates 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 [Mustafa M. Tikir 2004]. 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 [Buck and Hollingsworth 2004]. Buck *et al.* also compare different hardware mechanism for detecting uniprocessor memory hierarchy bottlenecks [Buck and Hollingsworth 2000b]. Satoh *et al.* study data-flow techniques to analyze data sharing patterns at compile time for OpenMP programs [Sato et al. 1999]. 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 [Thiffault et al. 2003]. We, in contrast, promote hardware-assisted sampling due to overheads resulting from software instrumentation in general.

11. CONCLUSION

In this paper, we present two novel hardware-assisted approaches to determine cache coherence bottlenecks. Our first method, *PMU-based tracing*, 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. 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 also describe another hardware-assisted method, *targeted tracing* that provides more control on the tracing process. With targeted tracing, we first use the PMU lossy load tracing feature to cut down on the number of instructions to instrument. The reduced set of instructions is instrumented using software instrumentation. Each load instance is timed, and loads are captured only if they cross a software defined latency threshold. Store instructions are sampled with different sampling intervals.

We evaluated both methods with a large set of OpenMP benchmarks and explored the tradeoffs between accuracy and overhead in terms of trace sizes and run-time slowdown. These approaches provide a low runtime overhead to identify coherence bottlenecks in OpenMP applications. PMU-based tracing has two possible sources of inaccuracy: coherence misses omitted due to sampling and the omission of a store that actually causes a coherence miss. Further, due to the lossiness of the trace, this method had a larger number of false positives when comparing false-sharing invalidations. With targeted tracing, the lossiness of load tracing is removed, and we experiment with different sampling intervals for tracing stores. In addition, we also characterized the accuracy of both these methods with respect to true-sharing and false-sharing invalidations. We found a weakness of reduced-trace methods for certain programs when evaluating false-sharing invalidations and suggested possible solutions to resolve it.

We show that both our methods reduce the number of loads captured by over two orders

of magnitude over full tracing. PMU-based tracing is more than an order of magnitude faster than full tracing and has high accuracy on most benchmarks. Targeted tracing provides even higher accuracy and control over the tracing process at the cost of relatively more overhead compared to PMU-based tracing.

Acknowledgments

This work was supported in part by NSF grants CAREER CCR-0237570, CNS-0406305, CCF-0429653, CNS-0410203 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-JC-151497. A preliminary version of this paper appeared in the the International Conference of Supercomputing, 2005 [Marathe et al. 2005].

REFERENCES

2002. The ASCI purple benchmarks. http://www.llnl.gov/asci/purple/benchmarks.

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

BAILEY, D. H., BARSZCZ, E., BARTON, J. T., BROWNING, D. S., CARTER, R. L., DAGUM, D., FATOOHI, R. A., FREDERICKSON, P. O., LASINSKI, T. A., SCHREIBER, R. S., SIMON, H. D., VENKATAKRISHNAN, V., AND WEERATUNGA, S. K. 1991. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications 5,* 3 (Fall), 63–73.

BREWER, E. A., DELLAROCAS, C. N., COLBROOK, A., AND WEIHL, W. E. 1992. Proteus: A high-performance parallel-architecture simulator. In *Proceedings of the SIGMETRICS and PERFORMANCE '92 International Conference on Measurement and Modeling of Computer Systems*. ACM Press, New York, NY, USA, 247–248.

BUCK, B. AND HOLLINGSWORTH, J. 2000a. An API for runtime code patching. *The International Journal of High Performance Computing Applications 14,* 4 (Winter), 317–329.

BUCK, B. AND HOLLINGSWORTH, J. 2000b. Using hardware performance monitors to isolate memory bottlenecks. In *Supercomputing*, ACM, Ed. 64–65.

BUCK, B. R. AND HOLLINGSWORTH, J. 2004. Data centric cache measurement on the intel itanium 2 processor. In *Supercomputing*, ACM, Ed.

BURGER, D., AUSTIN, T. M., AND BENNETT, S. 1996. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, University of Wisconsin, Madison. July.

DAVIS, H., GOLDSCHMIDT, S. R., AND HENNESSY, J. 1991. Multiprocessor simulation and tracing using tango. In *Proceedings of the 1991 International Conference on Parallel Processing*. Vol. II, Software. CRC Press, Boca Raton, FL, II–99–II–107.

DEROSE, L., EKANADHAM, K., HOLLINGSWORTH, J. K., AND SBARAGLIA, S. 2002. SIGMA: A simulator infrastructure to guide memory analysis. In *Supercomputing*.

GIBSON, J. 2003. Memory profi ling on shared memory multiprocessors. Ph.D. thesis, Stanford University.

HUGHES, C., PAI, V., RANGANATHAN, P., AND ADVE, S. 2002. Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors. *IEEE Computer 35,* 2 (February), 40–49.

INTEL. 2004. *Intel Itanium2 Processor Reference Manual for Software Development and Optimization*. Vol. 1. Intel.

Intel Corp. 2004. *Intel Itanium2 Processor – Reference Manual*. Intel Corp.

KRISHNAMURTHY, A. AND YELICK, K. 1995. Optimizing parallel programs with explicit synchronization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 196–204.

LEBECK, A. AND WOOD, D. 1994. Cache profi ling and the SPEC benchmarks: A case study. *Computer 27,* 10 (Oct.), 15–26.

LEBECK, A. AND WOOD, D. 1997. Active memory: A new abstraction for memory system simulation. *ACM Transactions on Modeling and Computer Simulation 7,* 1 (Jan.), 42–77.

LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V., AND HAZELWOOD, K. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

MARATHE, J., MUELLER, F., AND DE SUPINSKI, B. 2005. A hybrid hardware/software approach to effi ciently determine cache coherence bottlenecks. In *International Conference on Supercomputing*. 21–30.

MARATHE, J., MUELLER, F., MOHAN, T., DE SUPINSKI, B., MCKEE, S., AND YOO, A. 2003. METRIC: Tracking down ineffi ciencies in the memory hierarchy via binary rewriting. In *International Symposium on Code Generation and Optimization*. 289–300.

MARATHE, J., NAGARAJAN, A., AND MUELLER, F. 2004. Detailed cache coherence characterization for openmp benchmarks. In *International Conference on Supercomputing*. 287–297.

MARTONOSI, M., GUPTA, A., AND ANDERSON, T. 1992. Memspy: analyzing memory system bottlenecks in programs. In *Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*. 1–12.

MELLOR-CRUMMEY, J., FOWLER, R., AND WHALLEY, D. 2001. Tools for application-oriented performance tuning. In *International Conference on Supercomputing*. 154–165.

MOHAN, T., DE SUPINSKI, B. R., MCKEE, S., MUELLER, F., YOO, A., AND SCHULZ, M. 2003. Identifying and exploiting spatial regularity in data memory references. In *Supercomputing*.

MUSTAFA M. TIKIR, J. H. 2004. Using hardware counters to automatically improve memory performance. In *Supercomputing*, ACM, Ed.

NETHERCOTE, N. AND SEWARD, J. 2003. Valgrind: A program supervision framework. In *Proc. of the 3rd Workshop on Runtime Verification* (Boulder).

NGUYEN, A.-T., MICHAEL, M., SHARMA, A., AND TORRELLAS, J. 1996. The augmint multiprocessor simulation toolkit: Implementation, experimentation and tracing facilities. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE Computer Society, Washington - Brussels - Tokyo, 486–491.

PATIL, H., COHN, R., CHARNEY, M., KAPOOR, R., SUN, A., AND KARUNANIDHI, A. 2004. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *37th International Symposium on Microarchitecture*.

ROSENBLUM, M., HERROD, S. A., WITCHEL, E., AND GUPTA, A. 1995. Complete computer system simulation: The SimOS approach. *IEEE parallel and distributed technology: systems and applications 3*, 4 (Winter), 34–43.

SATO, M., SATOH, S., KUSANO, K., AND TANAKA, Y. 1999. Design of OpenMP compiler for an SMP cluster. In *EWOMP '99* (Lund). 32–39.

SATOH, S., KUSANO, K., AND SATO, M. 2001. Compiler optimization techniques for openMP programs. *Scientific Programming 9*, 2-3, 131–142.

TAO, J., SCHULZ, M., AND KARL, W. 2003. A simulation tool for evaluating shared memory performance. In *Proc. of the 36th Annual Simulation Symposium* (Orlando).

TAO, J. AND WEIDENDORFER, J. 2004. Cache simulation based on runtime instrumentation for OpenMP applications. In *Proc. of the 37th Annual Simulation Symposium* (Arlington, VA). 97–103.

THIFFAULT, C., VOSS, M., HEALEY, S. T., AND KIM, S. W. 2003. Dynamic instrumentation of large-scale mpi/openmp applications. In *International Parallel and Distributed Processing Symposium*.

WOLF, M. E. AND LAM, M. 1991. A data locality optimizating algorithm. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 30–44.

Appendix

Table IV shows the number of loads and stores logged and the time for tracing (in seconds). `FULL` denotes full access tracing, `Targeted` is the targeted tracing scheme and `PMU` is our hardware assisted lossy tracing scheme. For targeted tracing, the timing threshold was 64 cycles, and the store sampling probability was 1.0 (*i.e.*, log all stores). For the PMU-based tracing, the cycle threshold was 8 cycles, and the sampling interval was 1 (OV-1).

The values from the table were used to derive the normalized values shown for execution time (Figures 8 and 12), number of loads traced (Figures 5(a) and 9(a)), and the number of stores traced (Figures 5(b) and 9(b)). The table only notes the values for OV-1 (PMU-Tracing) and SMPL-1 (Targeted Tracing). The actual values for the other sampling

Table IV. Comparison of Access Traced, and Time for Tracing

Benchmark	# Loads (Millons)			# Stores (Millons)			Tracing Time (Seconds)		
	Full	Targeted	PMU	Full	Targeted	PMU	Full	Targeted	PMU
BT	399	2.30	3.57	92.75	115.84	10.21	315.64	117.00	15.84
CG	113	0.50	0.96	4.91	4.91	0.54	75.44	45.76	2.03
FT	46	0.16	0.89	16.47	16.47	1.67	39.03	16.24	2.36
LU	91	0.32	0.87	29.53	29.53	3.18	90.69	36.00	4.57
SMG2K	32	1.04	3.53	7.55	7.55	0.80	52.80	27.00	8.12
SP	88	0.98	1.40	28.54	28.54	2.91	86.60	50.52	5.28
SPPM	518	2.41	2.98	167.06	167.06	17.50	332.25	167.67	24.60
MG	100	0.39	0.23	9.99	9.33	0.96	66.30	28.78	13.21

intervals can be derived using this information in conjunction with the normalized values shown in the respective figures.

Received August 2005; revised March 2006; accepted July 2006