Source-Code Correlated Cache Coherence Characterization of OpenMP Benchmarks

Jaydeep Marathe and Frank Mueller, *Senior Member, IEEE*

***Abstract*— Cache coherence in shared memory multiprocessor systems has been studied mostly from an architecture viewpoint, often by means of aggregating metrics. In many cases, aggregate events provide insufficient information for programmers to understand and optimize the coherence behavior of their applications. A better understanding would be given by source-code correlations of not only aggregate events but also finer-granularity metrics directly linked to high-level source code constructs, such as source lines and data structures.**

In this paper, we explore a novel *application-centric* approach to studying coherence traffic. We develop a coherence analysis framework based on incremental coherence simulation of actual reference traces. We provide tool support to extract these reference traces and synchronization information from OpenMP threads at run-time using dynamic binary rewriting of the application executable. These traces are fed to ccSIM, our cache-coherence simulator. The novelty of ccSIM lies in its ability to relate low-level cache coherence metrics (such as coherence misses and their causative invalidations) to high-level source code constructs including source code locations and data structures. We explore the degree of freedom in interleaving data traces from different processors and assess simulation accuracy in comparison to metrics obtained from hardware performance counters.

Our quantitative results show that: (a) Cache coherence traffic can be simulated with a considerable degree of accuracy for SPMD programs, as the invalidation traffic closely matches corresponding hardware performance counters. (b) Detailed high-level coherence statistics are very useful in detecting, isolating and understanding coherence bottlenecks. We use ccSIM with several well known benchmarks and find coherence optimization opportunities leading to significant reductions in coherence traffic and savings in wall clock execution time.

***Index Terms*— Cache memories, simulation, dynamic binary rewriting, program instrumentation, SMPs, coherence protocols**

I. Introduction

High-performance computing platforms are increasingly deployed in configurations of multiprocessor shared-memory nodes. Understanding the coherence behavior of multi-threaded programs on such systems can lead to optimizations with significant impact on the overall wall-clock execution time of the program. Past work on understanding cache coherence has concentrated on two distinct areas: architecture simulation and program analysis for performance tuning. Many architecture and system simulators have been reported, supporting different coherence models (*e.g.*, [1], [2], [3], [4], [5], [6]), and they operate at varying levels of abstraction ranging from cycle accuracy to discrete event based simulation. In the performance tuning area, work has been focused mostly on compiler analysis to derive optimized code (*e.g.*, [7], [8]).

Hardware performance monitors of modern processors offer new opportunities for low overhead measurement of coherence activities. Here, we explore a complementary scheme where programmers use hardware counters to confirm that a potential coherence bottleneck exists in the program, and then use our framework to generate detailed source-code related information to understand its cause.

In this paper, we focus on a discrete event-based cache coherence simulation without cycle accuracy or instruction-level simulation. We constrain ourselves to an SPMD programming paradigm on dedicated SMPs. Specifically, we assume the absence of workload sharing, *i.e.*, only one application runs on a node, and we enforce a one-to-one mapping between threads and processors. These assumptions are common for high-performance scientific computing [9], [10].

ccSIM is the first tool to characterize coherence traffic for OpenMP programs. The novelty lies in being able to provide detailed per-reference source-code correlated statistics about coherence events (invalidations, coherence misses) and in showing how such tools can be used to detect, understand, and fix inefficiencies in accessing shared data in large well known benchmarks that closely resemble real world programs. In contrast to most previous approaches, ccSIM does not require any special compiler or linker support. It operates directly on the program executable and potentially allows the collection of *partial* access traces by toggling the instrumentation at run-time (dynamic instrumentation).

Our contributions are as follows: 1) We introduce `ccSIM`, a cache coherence simulator that we have designed and built for shared memory multiprocessors. 2) We develop a novel dynamic binary-rewriting mechanism to extract memory access traces and thread synchronization information from OpenMP parallel programs. 3) We demonstrate good correlation between ccSIM results and hardware performance counters for an SMP architecture on a variety of OpenMP benchmarks. 4) We quantify the run-time overhead of software instrumentation and evaluate several on-line compression algorithms with

Manuscript received May 9, 2006; revised October 25, 2002; accepted November 6, 2006.

Author's address: Dept. of Computer Science, North Carolina State University, Raleigh, NC 27695-7534, mueller@cs.ncsu.edu, phone: (919) 515-7889.

A preliminary version of this paper appeared in the International Conference on Supercomputing, 2004. This work was supported in part by NSF CAREER grant CCR-0237570 and through the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under subcontract # B540203. This research was supported in part by NSF cooperative agreement SCI-9619020 through resources provided by the San Diego Supercomputer Center.

respect to compression factors and execution time. 5) Finally, we demonstrate how ccSIM obtains detailed information indicating causes of invalidations and coherence misses and relates these events to their program location and data structures. We achieve significant wall-clock time improvements for several well known benchmarks by inferring optimization opportunities from the information supplied by ccSIM.

II. ccSIM Framework

Figure 1 shows the ccSIM framework. There are 3 phases in our approach - *Instrumentation*, *Trace generation* and *Coherence simulation*. First, the target OpenMP executable is instrumented for capturing the memory access trace and OpenMP synchronization information. During execution, the instrumentation calls handler functions in a shared library that compress the event trace and write the compressed representation to stable storage. An incremental shared memory multiprocessor simulator uses this event trace to simulate coherence traffic for a selected coherence protocol. The simulator maps the coherence events (*e.g.*, invalidations, coherence misses) to high-level constructs, such as source code locations and also to local and global variable names. The simulator achieves this using the symbolic information extracted from the target OpenMP executable by the instrumenter (controller) program. At the end of simulation, the detailed coherence metrics are presented to the user. In our work, we explicitly bind each OpenMP thread to a different processor using the `bind_processor` system call. Thus, the per-thread event trace is actually a per-processor event trace. Each phase is discussed in more detail in the following.

A. Instrumentation

Our instrumentation tool uses the DynInst instrumentation library [11] for dynamic program instrumentation. It is an extension of our previous work in using binary rewriting to extract memory traces from uniprocessor programs [12].

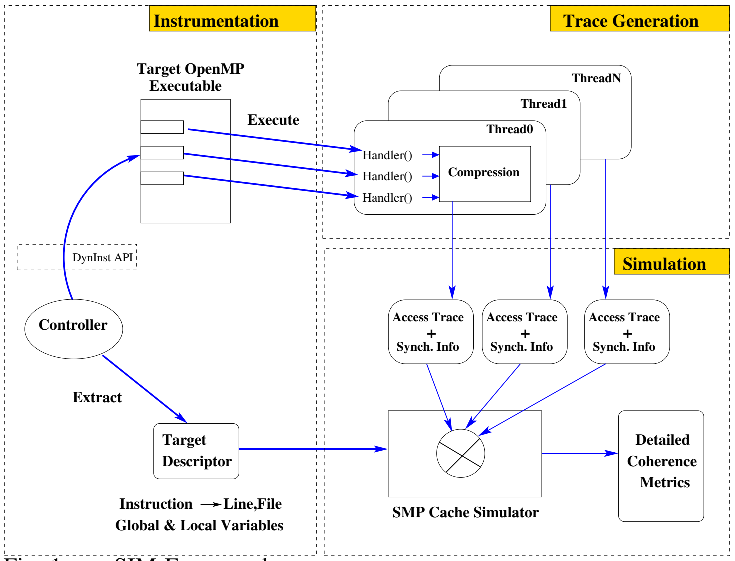


Fig. 1. ccSIM Framework

In this work, we extend the original tool to support multi-threaded OpenMP programs.

The instrumentation process occurs as follows. A control program (controller) attaches to the potentially executing target OpenMP program. For each OpenMP thread, the controller inserts instrumentation to intercept the memory access instructions (loads and stores). To reduce the overhead of trace collection, the controller does not instrument instructions that access memory locations at an offset from the stack pointer register. These memory instructions access stack locations that are private to each thread (since each thread has a separate stack). It is uncommon that a thread's stack variables will be accessed by other threads such that exclusion of such instructions during instrumentation will not result in any measurable loss of accuracy. In addition, we also instrument the compiler-generated functions that implement OpenMP synchronization constructs (*e.g., #pragma parallel do, #pragma barrier, etc.*). This synchronization information is saved in the captured event trace. During simulation (phase 3), the synchronization information allows us to maintain a correct ordering among accesses from different threads (*e.g.*, no accesses from any thread past a barrier can be simulated till all accesses from all threads before the barrier have been simulated). Finally, the instrumentation also records function entry and exit events, as well as the stack base address when the function was entered. The former allows us to tag coherence traffic to specific functions. The latter allows us to also support tagging coherence traffic to local variables whose addresses are not determined till the function is entered. [1]

To support tagging of coherence events to high level constructs, the controller extracts symbolic information from the target executable. This symbolic information is embedded in the target executable.[2] This information is used to map the memory access instructions to locations in the source code (line::File). In addition, the names and addresses of global variables as well the names and stack offsets of local variables for each function are extracted and stored in a target descriptor file.

B. Trace Generation

The instrumentation instructions call handler functions in a shared library that is loaded into the target program's address space using a one-shot instrumentation. Once the instrumentation is complete, the target program is allowed to continue execution. As the program executes, the handler functions get invoked, generating an event trace (memory accesses, function entry/exits and OpenMP synchronization

[1]The debug information embedded in the executable contains the *offset* values for each local variable of a function. The offset values can be combined with the value of the stack pointer to get the absolute memory address of the local variable for that instance of the function.

[2]Most compilers support inserting debug information in the binary, *e.g.*, with the `-g` flag.

calls). For real-world programs, the tool can be expected to capture hundreds of millions of events. To conserve space, it is essential to efficiently compress this trace *online* before storing it to stable storage. In later sections, we discuss and evaluate several compression strategies.

Our instrumentation framework allows *partial* event tracing. After an adequate number of events have been captured, the instrumentation can be turned off, and the original application can continue execution without any instrumentation overhead. This is important for tracing "snippets" of long-running applications. In this paper, however, we only collect *full* event traces, *i.e.*, we run the application from start to finish and use the generated event trace for processing.

Each thread is responsible for logging its own event trace. There is no cross-thread dependence for tracing. Hence, our framework scales with increasing number of threads.

C. Simulation

This is the final phase. The simulator uses the compressed per-thread event trace for incremental coherence simulation. In this work, ccSIM simulates the MESI coherence protocol that is present on our target platform. Other protocols can be easily simulated, if required in the future.

Interleaving of Reference Streams: It is important to note that for correct coherence simulation, we must not only capture the memory access trace but also the partial ordering information among the OpenMP threads. The partial ordering among threads occurs due to the execution of OpenMP synchronization directives, *i.e.*, `barriers`, `critical` sections, `atomic` sections and accesses protected by explicit mutex locks (`omp_get_lock`, `omp_set_lock`).

We maintain the partial ordering during simulation in the following manner. In the instrumentation phase, we instrument the entry and exit points of the functions implementing the OpenMP directives in the compiler's run-time support library. These recorded events are used to *order* accesses from different threads during coherence simulation. For barrier events, the simulator ensures that all events from all threads before the barrier are executed before any events after the barrier. The mutual exclusion effect of `critical`, `atomic`, `omp_get_lock()` and `omp_set_lock()` directives is achieved by allocating and manipulating corresponding lock structures in the simulator.

For understanding coherence behavior more effectively, we found that it is useful to classify accesses within and across a *region*. We define a *region* as the execution between two successive barrier events.[3] In a region without additional OpenMP synchronization events (*e.g.*, `omp critical`), there *is no ordering between accesses from different threads*. We explore the effect of different interleavings by allowing our simulator to execute in two modes at the start of a region:
Interleaved Mode: The simulator processes one data reference from each trace (corresponding to a thread or processor) before processing the second reference for each trace etc. Effectively, the simulator enforces a fine-grained interleaving in a round-robin fashion on a per-reference base in this mode.
Piped Mode: The simulator processes all data references from one trace up to the next synchronization point before processing data references from the second trace etc., effectively enforcing a coarse-grained interleaving at the level of regions.

A comparison of results from the interleaved and piped modes reflects the extent to which program latency is affected by the non-deterministic order of execution of OpenMP threads and may provide extremes (bounds) on metrics for coherence traffic.

Example: Figure 2 shows the trace events and simulator actions for a simple OpenMP program with two active OpenMP threads. `A` and `B` are shared arrays of size N, and `i` is a local variable. Static loop scheduling is assumed for the OpenMP `for` loop. The entry into the parallel OpenMP region is logged as a trace event and causes the simulator to activate two driver objects. Accesses generated by each OpenMP thread to the `A` and `B` arrays are logged separately. The drivers may simulate these accesses in parallel, as shown for the interleaved mode. When an OpenMP thread exits from the implicit barrier at the end of the `for` loop, a `barrier exit` event is logged for that thread. Detection of a barrier event causes drivers to synchronize. Another synchronization takes place when the `parallel end` event is processed. After an OpenMP parallel region, a serial phase starts, and only one driver (corresponding to the master thread) will remain active. All others remain unused till the start of the next parallel phase.

D. Studying Invalidations and Misses

A key metric for the identification of memory performance bottlenecks in a multiprocessor system is the number of invalidations to lines in the lowermost level of cache of each processor. Invalidations cause coherence traffic, thereby increasing the utilization of the shared bus in a symmetric multiprocessor architecture. More significantly, these invalidations could lead to coherence misses. Since coherence misses will miss in all levels of cache (the data being accessed is in a modified state in some other processor's cache), the latency for the miss will be high and contemporary out-of-order superscalar processors would stall till the miss is satisfied (since the out-of-order window has limited size). Thus, reducing the volume of coherence misses often has a direct impact on the overall wallclock execution time.

[3]Our definition of region is slightly different from its definition in the OpenMP 2.5 standard. Even though both definitions refer to the *dynamic* extent of execution, our focus is only on barrier events. In contrast, the OpenMP standard defines regions more generally as the dynamic or runtime extent of a *construct* or OpenMP library routine [13].

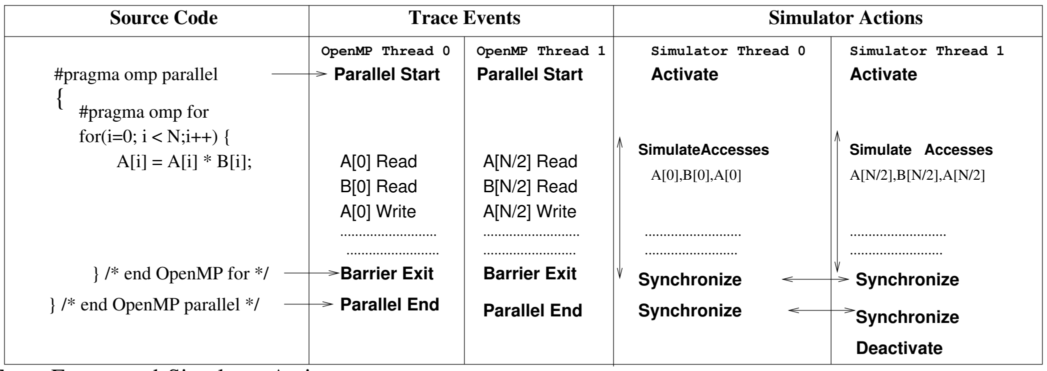


Fig. 2. Illustration: Trace Events and Simulator Actions

Since the main motivation in reducing the invalidate traffic is to decrease the number of coherence misses, it is imperative to distinguish between coherence misses and uniprocessor misses in a processor. Invalidations to cache lines can further be classified as true-sharing invalidations and false-sharing invalidations in each level of cache. True-sharing invalidations arise from accesses to the same shared memory location by more than one processor, with at least one access being a write access. False-sharing invalidations are caused due to accesses to different memory locations that map to the same cache line on more than one processor. ccSIM maintains state between accesses to a cache line to detect and distinguish true/false sharing. We introduced the concept of a *region* above (Section II-C). Invalidations can be classified as *in-region* and *across-region* as shown in Figure 3. Within the same region, we further distinguish true-sharing invalidations as follows: 1) References not protected by locks: These typically occur in the single-writer, single/multiple-reader scenario where one processor writes to a common location and one or more processors read from it. If there are multiple processors that write to a shared memory location, it may indicate the existence of a data race condition in the program. In this case, our tool will pinpoint the exact source code references involved in the race. We found such a data race using our tool, in the ASCI Purple benchmark sPPM (Section IV-E). 2) References protected by locks: These typically occur in the multiple-writer, single/multiple-reader scenario where multiple processors write and read from a common location.

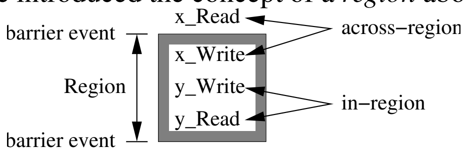


Fig. 3. Classification of Invalidations

In summary, ccSIM generates the following metrics 1) **Uniprocessor statistics**: Hits, misses, temporal and spatial locality ratios, and list of evictors for each reference. The uniprocessor metrics are described in our previous work [12]. 2) **Invalidations**: These are sub-classified into *true* and *false*-sharing invalidations, as discussed above. 3) **Coherence Misses**: A miss is classified as a coherence miss if it is accessing data that was present in the processor's cache previously but was invalidated due to a write from another processor to the shared data's memory line. When a cache line is invalidated, we save the cache tag of the invalidated line. Later, when a miss occurs, this information is used to classify a miss as a coherence miss. 4) **Invalidator Lists**: We have enhanced our framework described in [14] to generate *invalidator lists* for each reference. The invalidators for a reference are the write (store) references on other processors that invalidated the data accessed by this reference. Invalidator lists help to understand the movement of shared data elements across processor caches. A later case study (ammp) shows the use of these lists for understanding coherence patterns.

These statistics can be viewed at several levels of detail: 1) **Per-Processor**: This level of detail is similar to architecture-oriented coherence simulators. 2) **Per-Reference**: A source code reference is a program location (line:File). Per-reference results allow us to magnify per-processor results and to map them to individual program locations. 3) **Per-Function**: Since we instrument function entry and exit points, we can generate per-function as well as per-calling context coherence metrics. 4) **Per-Variable**: Global and local variables are supported by our framework. In addition, dynamically allocated variables can be distinguished by their call-context-sensitive allocation site in the program source code. 5) **Within/Across OpenMP regions**: As discussed before, we distinguish between interactions that occur in the same OpenMP region from interactions that occur across different OpenMP regions.

The coarser levels of detail can be used to quickly check whether a potential coherence bottleneck exists (*e.g.*, high ratio of coherence misses to total misses). Then, the per-reference and per-data structure metrics can be used to isolate the bottleneck to particular source code locations and data structures. Finally, the invalidator lists show how the shared data is moving across processor caches. We demonstrate this performance evaluation process with several case studies.

III. EXPERIMENTS

First, we present the OpenMP benchmarks used for experiments with ccSIM. Next, we compare results obtained from ccSIM with hardware performance counters. We evaluate the trace extraction framework with respect to execution overhead induced on the target application and compare the effectiveness of various compression strategies for online compression of the access stream.

Finally, we use ccSIM to characterize the shared memory usage of representative OpenMP benchmarks and show how ccSIM statistics are useful in detecting and isolating coherence bottlenecks.

Benchmarks: In later sections, we validate our simulator against hardware performance counters and measure the overhead of tracing with different compression algorithms. For these experiments, we selected the 6 benchmarks from the NAS OpenMP suite [15] plus an additional OpenMP benchmark (NBF) from the GROMOS benchmark suite [16].

A brief description of each benchmark is given below. 1) IS: A large integer sort used in "particle method" codes. 2) MG: A V-cycle MultiGrid method to compute the solution of the 3-D scalar Poisson equation. 3) CG: A Conjugate Gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix. 4) FT: An implementation of a 3-D Fast Fourier Transform (FFT)-based spectral method. 5) SP: A simulated CFD application with scalar pentagonal bands of linear equations that are solved sequentially along each dimension. 6) BT: A simulated CFD application with block tridiagonal systems of 5x5 blocks solved sequentially along each dimension. 7) NBF (Non-Bonded Force Kernel): A molecular dynamics simulation computing non-bonded forces due to molecular interactions.

All NAS benchmarks used class S data sets, except for IS which used class W. The NBF kernel was run for 2 time steps with 16384 molecules. For these settings, we observed a sufficient number of invalidations to characterize the application behavior.

In addition, we also present case studies in using ccSIM to optimize much larger applications, which closely resemble real world programs. These include two benchmarks (IRS-1.4, SMG2000) from the ASCI Purple OpenMP suite [17], and one benchmark (AMMP) from the SPEC2001M OpenMP suite [18]. More details about these applications are presented in the case studies.

A. Comparison with Hardware Counters

In this section, we validate ccSIM against measurements from hardware performance counters. From a developer's perspective, the number of *coherence misses* is the most important facet of the shared memory access pattern of an application. However, there are no hardware counters capable of measuring coherence misses on our target platform. Instead, we compare the number of *invalidations* for ccSIM against the actual number of invalidations measured by the hardware counters. The total number of invalidations is an upper bound on the number of coherence misses for the application. Reducing invalidations will also lower the number of coherence misses, thereby improving application performance.

Hardware Environment: The hardware counter measurements were carried out on a 4-way SMP machine with 375 MHz Power3 processors. The hardware counters were accessed through the proprietary `Hardware Performance Monitor (HPM)` API. The system has a 64 KB 128-way associative L1 cache with round-robin replacement and an 8 MB 4-way associative L2 cache. All experiments were carried out with 4 active OpenMP threads bound to distinct processors. The IBM OpenMP compilers, xlc_r and xlf_r, were used to compile the benchmarks at the default optimization level O2 with following flags settings: `-qarch=auto, -qsmp=omp, -qnosave`.

HPM measurements: The Power3 hardware implements the MESI coherence protocol within an SMP node. The `PM_SNOOP_L2_E_OR_S_TO_I` and `PM_SNOOP_M_TO_I` HPM events were used to measure the number of L2 cache invalidations with E→I, S→I and M→I transitions, respectively. The OpenMP runtime system also contributes to the number of invalidations measured. Since we are interested only in the invalidations of the application, we need to remove these invalidations from the measured numbers.

To assess the side-effect of the OpenMP runtime system on invalidations, we measured invalidations for OpenMP runtime constructs with empty bodies in a set of microbenchmarks. For example, the overhead in terms of invalidations for a barrier construct was determined. The microbenchmarks were subsequently used to adjust raw HPM data obtained from application runs by removing the extrapolated effect of OpenMP runtime invalidations for n iterations. For example, we removed the effect of $n = 100$ times the overhead for a single barrier if the benchmark contained 100 barriers. We refer to these measurements as the *raw* HPM metrics and the *OpenMP-adjusted* HPM metrics.

Table I shows the raw and OpenMP-adjusted HPM measured invalidations for the L2 cache. The invalidations were measured for each processor separately using the HPM events discussed above and summed up to get the total invalidations shown in the table. Each HPM measurement is the mean of 5 samples.

Comparison with ccSIM: ccSIM was configured with the MESI coherence protocol and with the cache parameters of the hardware platform (4-way Power3 SMP node). Both L1 and L2 caches were simulated. Table II compares total L2 invalidations for HPM and the two ccSIM modes - *piped* and *interleaved*.

The results indicate a good correlation between ccSIM and HPM for most benchmarks. The absolute error between ccSIM and HPM is less than 17% for all benchmarks and less than 7% for most. Moreover, for the NAS benchmarks, both interleaved and piped modes result in closely matching

TABLE I
TOTAL L2 INVALIDATIONS WITH HPM

Benchmark	HPM(raw)	HPM(OpenMP-adjusted)
IS	165246	162964
MG	24631	13629
CG	134964	100488
FT	326595	325257
SP	282269	258923
BT	185317	157384
NBF	474121	135926

TABLE II
HPM VS. CCSIM

Benchmark	HPM	ccSIM	ccSIM	% Error
		Interleaved	Piped	Interleaved vs. HPM
IS	162964	163073	159913	-0.06
MG	13629	13174	12355	3.30
CG	100487	117117	116318	-16.50
FT	325257	302630	302607	6.90
BT	157384	157503	157480	-0.07
SP	258922	268334	268334	-3.60
NBF	135926	137498	14629	-1.15

numbers of invalidations. This indicates that for these benchmarks, fine-grained round-robin simulation is not necessary to achieve a high level of simulation accuracy. NBF stands out as an anomalous case with significant difference between the interleaved and piped modes of simulation. ccSIM allows us to categorize invalidations into true and false sharing invalidations as well as to distinguish between across-region and in-region invalidations, as explained in Section II-D. The cause of the discrepancy becomes apparent when we examine the in-region true-sharing critical invalidations shown in Figure 4. Metrics are plotted in a log scale. The number of true-share invalidations occurring within a region is much higher (at least an order of magnitude) in the interleaved simulation mode. The interleaved simulation mode involves fine-grained round-robin simulation, which leads to a "ping-pong" exchange of shared data across processors. The ping-pong exchange does not take place with the piped mode of simulation, leading to a very small number of invalidations to be recorded. A look at the per-reference ccSIM statistics indeed shows that the most significant invalidation source is a data access point inside an OpenMP `critical` construct. This demonstrates the necessity of interleaved simulations for codes containing critical sections to closely resemble the interleaving of references during actual execution.[4]

B. Execution Overhead and Trace Compression

Instrumentation for capturing the memory access trace imposes execution overhead on the application. The access traces being captured can be in the order of hundreds of gigabytes. Hence, effective compression is necessary before they can be stored to disk. In this section, we measure the run-time overhead imposed by software instrumentation. We also evaluate several compression strategies with respect to additional run-time overhead imposed and the quantum of compression achieved by each.

[4]In Figure 4, the number of in-region true-sharing invalidations is shown to be zero for P4. This is an artifact of our round-robin scheduling due to which the true-sharing invalidations were classified as across-region false-sharing invalidations in this particular benchmark. This can potentially be improved upon by using pseudo-random instead of round-robin scheduling, after which the results for P4 will be similar to other processors.

For compression, we compare the following strategies: 1) **No Compression (No-Compr)**: No compression algorithm is used. The raw uncompressed trace is written to stable storage. 2) **PRSD Compression (PRSD-Compr)**: This compression algorithm is targeted for regular accesses in nested loop structures, as commonly found in scientific programs. It is reported in our previous work in [19]. 3) **LZO Compression (LZO-Compr)**: This is an open-source lossless compression library designed specifically for compression speed [20]. We use the `mini-lzo` variant that implements the LZO1X-1 algorithm. Compression input is in chunks of 64KB. 4) **Multi-stage Compression (Multi-Compr)**: This is a hybrid algorithm that uses LZO compression to compress the output stream of the PRSD algorithm.

Run-Time Overhead: Figure 5(a) shows the execution time of just the software instrumentation (`Null-Instru`), and for instrumentation plus compression with the algorithms discussed above. The execution time is normalized to the execution time of the original unmodified executable. We make the following observations: 1) The cost of software instrumentation alone (`Null-Instru`) is approximately 2 to 3 orders of magnitude (*i.e.*, 100 to 1000 times slowdown). This is due to the high frequency of instrumentation at every load and store instructions. 2) The execution overhead of storing the compression trace is comparatively low (`No-Compr` vs. `Null-Instru`). 3) LZO Compression is very fast and adds very little overhead by itself (`LZO-Compr`, `Multi-Compr`). 4) PRSD Compression has variable over-

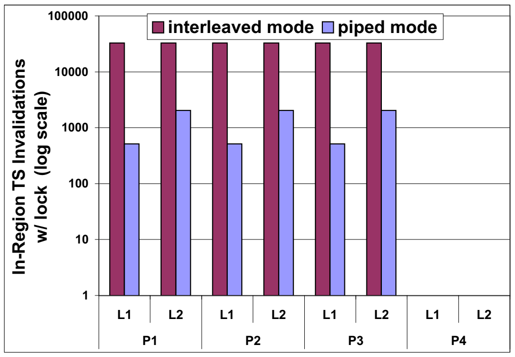
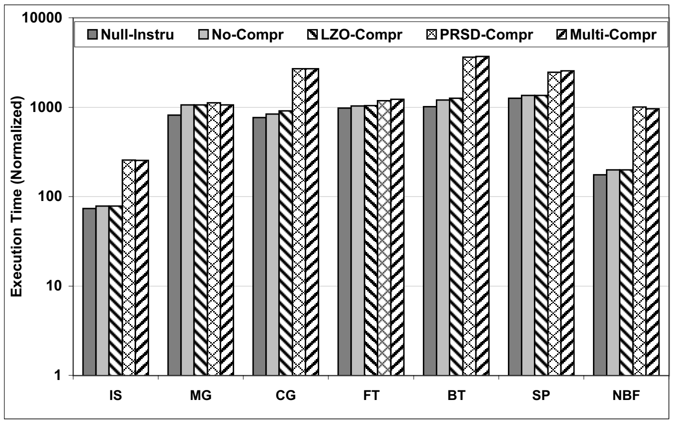
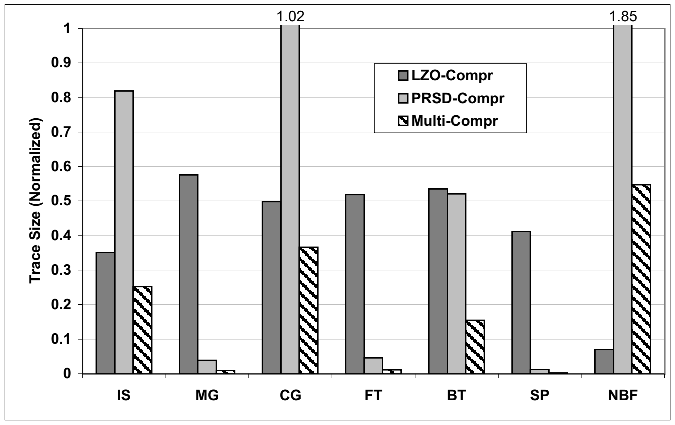


Fig. 4. NBF: Interleaved and Piped



(a) Execution Time (Normalized)



(b) Trace Size (Normalized)

Fig. 5. Execution Overhead and Trace Sizes

head. For some benchmarks (MG, FT, SP) the overhead is low while for others, there is significant overhead compared to LZO compression.

Trace Compression: Figure 5(b) compares the trace sizes achieved with the various compression strategies normalized to the original size of the trace. We make the following observations: 1) LZO compression always reduces the trace size by half or even more. 2) PRSD-based compression can lead to spectacular compression in some cases (MG, FT and SP) and beats LZO-based compression in 4 out of the 7 benchmarks (MG, FT, SP, BT). However, the compression rate is significantly better for LZO for the remaining three benchmarks (IS, CG, NBF). 3) Multi-stage compression achieves the best compression for all benchmarks, except for NBF. Even with NBF, multi-stage compression reduces the trace size to approximately half of the original size.

To summarize, PRSD-compression either works very well (with low execution overhead and very high compression) or very poorly (with relatively high overhead and poor compression). Compression is poor when the program either does not have nested loops that dominate the overall memory accesses or the access stream generated is irregular. The latter is the cause for the poor compression rate of both CG and NBF, due to the presence of indirectly indexed arrays in sparse matrix computations, which generate a non-linearly strided access stream.

A hybrid multi-stage algorithm (PRSD+LZO) almost always achieves the best compression, at the price of additional execution overhead.

IV. Opportunities for Transformations

In this section, we demonstrate how ccSIM is used to detect and isolate coherence traffic bottlenecks, to derive opportunities for transformations leading to reduced coherence traffic and, thereby, to obtain potential performance gains.

Our methodology for performance evaluation is subject to a cost/benefit trade-off, as detailed in the following. A high overhead of tracing and simulation limits the extent of the program execution that can be realistically traced by our framework. We expect the programmer to either create a smaller data set or to identify a repeating program phase (*e.g.*, a single timestep) for performance evaluation. The resulting smaller program trace must have similar sharing characteristics as the original one; otherwise, the performance analysis results may not apply to the original program. Consider a the smaller program's data set that completely fits in cache while the original program's data set does not. Then, the importance of coherence (sharing) optimization may be exaggerated by the performance analysis.

Tracing has relatively high overhead. Thus, we recommend that programmers follow a two-step approach for performance evaluation of coherence activity. First, existing hardware performance counters can be used to quickly and cheaply evaluate if there exists a significant amount of sharing between processor caches (*e.g.*, using our previous approaches [21]). If such sharing exists, then our framework can provide detailed source code level information about the causes of any potential sharing bottlenecks.

Except for NBF, all our case studies use a smaller data set for performance evaluation and the recommended large data set that is used for measuring performance improvements. We are able to effectively use smaller data sets due to two notable reasons. First, 3 out of the 4 use cases (NBF, SMG2000, AMMP) exhibit sharing behavior between processors that is *temporally close*. In other words, the same sharing behavior will occur for small or large data sets, irrespective of whether the data set fits in cache or not. Second, for all use cases, the coherence simulation results lead us to optimizations (removing redundant concurrency, increasing concurrency, prefetching) that provide performance benefits irrespective of whether the data set fits in cache or not. In the first case, this is a property of the trace while in the later one, it is a property of the optimizations. This shows that in practice, the potentially difficult task of crafting smaller data sets or truncated program runs that reflect the original program behavior may be mitigated.

We shall now use ccSIM to optimize the NBF kernel. This code is comparatively simple compared to the other applications that we discuss later (irs, smg, ammp). NBF serves as a good introduction to characterizing and optimizing coherence behavior with ccSIM, even though the code analysis and transformations we discuss for it are straightforward, and may be achievable by visual inspection of the code. The other benchmarks are much larger and complex, and a profile-guided approach (like our tool) would be essential to understand and optimize their coherence behavior.

A. NBF: Non-Bonded Force Kernel

A full access trace was obtained for the OpenMP NBF kernel. The OpenMP environment was set to four threads and static scheduling (`OMP_NUM_THREADS=4`, `OMP_SCHEDULE=STATIC`). **Analysis:** Figure 6 shows the breakdown of misses for L1 and L2 caches for each processor obtained by ccSIM.

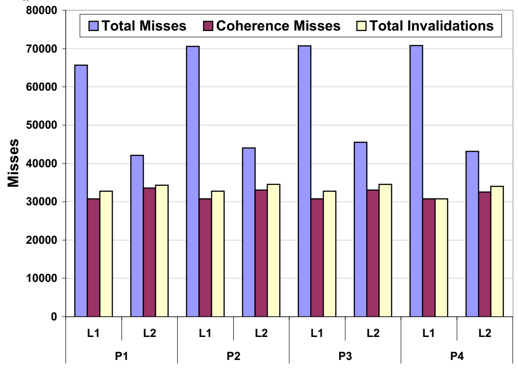


Fig. 6. NBF: Breakdown of L2 misses

We observe that almost all L2 misses and a significant number of L1 misses are coherence misses. A coherence miss is caused when a processor accesses a cache line that was invalidated due to a write from another processor. However, a large number of invalidations does not necessarily imply a large number of coherence misses, since the invalidated cache lines may not be referenced by the processor again before being flushed out of the cache. The number of coherence misses shown in Figure 6 is very close to the number of invalidations received by the cache. This shows that almost all invalidations eventually caused a coherence miss. Minimizing the total number of invalidations will also reduce the magnitude of coherence misses correspondingly.

We have detected a potential coherence bottleneck. We can use the per-reference coherence and cache statistics generated by ccSIM to determine the *cause* of the bottleneck. Table III shows the per-reference statistics on processor one for the top three references of the original code and two optimization strategies (serialized and round-robin) discussed next. Only L2 cache statistics are shown.

We observe that access metrics across all processors are uniform. The `f_Read` reference on line `141` of the source code has an exceptionally high miss rate in all processors. Moreover, more than 96% of the misses for this reference are coherence misses. The invalidation data shows that the large number of *in-region* invalidations are the primary cause for these misses. The relation of this reference to the source code indicates that line 141 is of interest:

```
#pragma omp parallel
...
for (i = 0; i < natoms; i++) {
  #pragma omp critical
141: f[i] = f[i] + flocal[i];
}
```

The for loop updates the global shared array `f` with values from the local private copy `flocal` for each OpenMP thread. The large number of invalidations attributed to the `f_Read` reference is due to a ping-pong exchange of the shared `f` array between processors as all of them try to update the global `f` array simultaneously.

Optimizing Transformations: Using ccSIM's per-reference statistics, we isolated the coherence bottleneck to the updates of the shared global array `f`. We shall discuss two ways of reducing the number of coherence misses. One method eliminates the ping-pong exchange of the `f` array by *serializing* the updates to the array `f` since they require mutually exclusive writes. This is achieved by moving the critical section to encompass the entire `for` loop instead of the single update. The modified code is shown below.

```
#pragma omp parallel
...
#pragma omp critical
for(i = 0; i < natoms; i++) {
  f[i] = f[i] + flocal[i];
}
```

Moving the `critical` statement outside the loop also reduces the number of times that the mutual exclusion region must be entered and exited, decreasing the execution overhead. Although reducing the number of coherence misses, this method does not exploit the potential for parallel updates to separate parts of the `f` array by different threads. Hence, we consider an alternate transformation. We can exploit parallelism by partitioning the array `f` into a number of segments. Each thread updates a distinct segment until all segments are updated. We call this scheme the *round-robin* update scheme. The modified code is shown below as pseudo-code.

```
//1. calculate #segments
tot_segments = (size of "f" array) / #threads;

//each thread executes this for loop
for(i = 0; i < tot_segments; i++) {
  //2. get segment id to update
  seg_id = (thread_id + i) % tot_segments;
  //3. update segment seg_id of array "f".
  ...
  //4. synchronize all threads (barrier)
```

TABLE III
NBF: COMPARISON OF PER-REFERENCE STATISTICS FOR EACH OPTIMIZATION STRATEGY

Line No.	Ref	Optimization Strategy	Misses	Miss Ratio	% Coherence Misses	Invalidations				
						Total	True		False	
							In Region	Across Region	In Region	Across Region
141	**f_Read**	**Original**	**32500**	0.99	**96.87%**	**32768**	**32768**	0	0	0
		Serialized	**2050**	1.0	**50.30%**	**2048**	2048	0	0	0
		Round-robin	**1790**	0.87	**42.84%**	**2048**	2048	0	0	0
227	**x_Read**	**Original**	1540	0.997	99.74%	768	1	765	0	2
		Serialized	1540	0.997	99.74%	768	1	765	0	2
		Round-robin	1540	0.997	99.74%	768	0	766	0	2
217	**f_Read**	**Original**	512	1.0	100%	256	256	0	0	0
		Serialized	512	1.0	100%	256	256	0	0	0
		Round-robin	512	1.0	100%	256	0	255	0	1

TABLE IV
NBF: WALL CLOCK TIMES (SECONDS)

Code Segment	Original	Serialized	Round-robin
f-Update	4.981	0.003 (**99.9%**)	0.003 (**99.9%**)
Other	2.141	2.076 (3%)	2.190 (-2.28%)
Overall	7.122	2.079 (**70.8%**)	2.193 (**69.2%**)

TABLE V
NBF: L2 INVALIDATIONS (HPM RAW)

Code Segment	Original	Serialized	Round-robin
f-Update	503654	921	6209
Other	37987	32916	38863
Overall	541641	33837	45072

```
  barrier();
}
```

Results: Table III compares the L2 coherence misses and invalidations for the two optimization strategies. Statistics are depicted only for processor-1 and are similar for the other processors. We observe that both strategies lead to a significant decrease in the volume of coherence misses for the `f_Read` reference. Table IV shows the wall clock execution time for (a) the routine that updates the shared array `f`, (b) the remainder and (c) the entire program. Table V shows the total L2 invalidations from all processors for each approach measured with HPM. We observe that the transformations cause a significant improvement in wall clock execution time. This improvement occurs due to two effects. First, the restructured programs have far less invalidations (and, subsequently, coherence misses) compared to the original program (Table V). Second, the restructured programs have lower OpenMP execution overhead because they execute fewer OpenMP calls.

B. IRS: Implicit Radiation Solver

IRS-1.4 is part of the ASCI Purple codes [17]. IRS can use MPI, OpenMP or a mixture of both for parallelization. We use the pure OpenMP version of IRS for our study. Existing OpenMP parallelization uses "`omp parallel do`" constructs for loop level parallelization. For the analysis below, we ran IRS for 10 calls to the top-level `xirs` function, with a limited data set (`NDOMS=10, ZONES_PER_SIDE=NDOMS_PER_SIDE`) with 4 OpenMP threads and static scheduling. This partial data trace is comparatively small, yet captures essential coherence traffic. Once our optimizations are complete, we compare the wall-clock time for the recommended full-sized data set for IRS (`zrad.008.seq`).

Analysis: Figure 9 shows that for all processors, coherence misses constitute almost the entire volume of L2 cache misses. Interestingly, the coherence miss magnitudes are asymmetric with processor-1 experiencing more than twice the number of coherence misses of any another processor. Figure 7 shows the per reference coherence statistics for processors 1 and 2. Statistics for other processors were similar to those for processor-2. References have been collected into groups with distinct coherence characteristics (Groups 1, 2 and 3). Multiple references are shown with only a single representative reference. For example, there are a set of fourteen references to different arrays in the `matrix` structure, all of which show similar coherence characteristics;

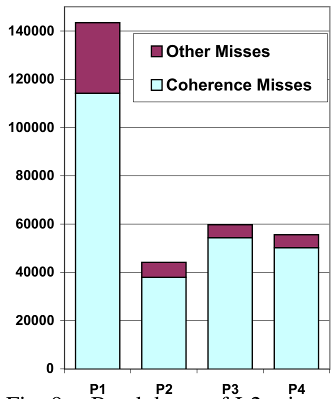


Fig. 9. Breakdown of L2 misses

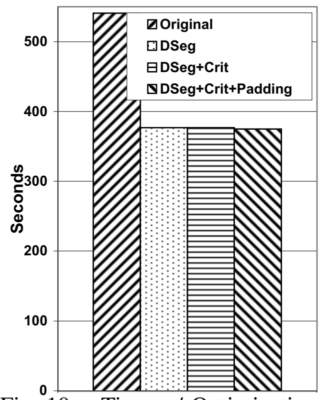


Fig. 10. Time w/ Optimizations

Fig. 7. IRS: Per-Reference Statistics

Proc	No.	Reference	Grp	Coh.	Invalidations				Optimization	Opt.
					True		False			Coh.
				misses	in	across	in	across	strategy	misses
1	1	v1[]_rd		8627	4	**7517**	31	1342	**code**	**1980**
	2	v2[]_rd		8568	310	**5093**	78	3085	**transforms**	**1971**
	3-16	matrix.dbl[]_wr	**1**	2547	25	**2325**	0	455	**for data**	**719**
	17	x[]_rd		1803	0	**1402**	391	2	**segregation**	**968**
	18	timersflag_rd	**2**	3182	1	0	**3122**	70	**padding**	**0**
	19	thread_flop[]_rd		1789	0	2	**1789**	0		**0**
	20	clock_last_rd	**3**	2165	**2166**	0	0	0	**remove sharing**	**0**
2	1	clock_last_rd	**3**	5997	**5644**	353	0	0		**0**
	2-3	timersflag_rd		2907	18	0	**2908**	0		**0**
	4-6	thread_flop[]_rd	**2**	2734	0	0	**2407**	327		**0**
	7	thread_wall_secs[]_rd		1022	0	0	**652**	371	**padding**	**0**
	8	thread_cpu_secs[]_rd		811	0	71	**742**	0		**0**

```
/* only master */
for (i =0; i < nblk; i++)
  dotprev += icdot(r[i], z[i],...);
    /* Reads r,z */
...
/* parallel updates to r,z */
#pragma omp parallel for
for (i =0; i < nblk; i++) {
  setpz1(r[i],...); /* Writes to r*/
  setpz1(z[i],...); /* Writes to z*/
}
...
/* only master */
for (i =0; i < nblk; i++)
  dotrz += icdot(r[iblk], z[iblk],...);
    /* Reads r,z */
```

Fig. 8. IRS Breakup into Parallel Regions

these are represented by a single representative reference `matrix.dbl[]` in the table. We observe that the set of references with significant coherence behavior are quite different for processor-1 and processor-2. We shall now analyze references belonging to each group in detail.

Group 1: These references account for the largest fraction of coherence misses. True sharing across-region invalidations are dominant for this group. This indicates that the data elements accessed by these references move across the L2 caches of multiple processors. Consider the first two references (`v1[]` and `v2[]`). These references occur in the `icdot` function, that is only called at three locations from the `MatrixSolveCG` function. All call sites are in serial code, *i.e.,* they are executed only by the master thread. Between successive calls, the argument arrays are updated by other processors in parallel regions, as depicted in Figure 8.

Thus parts of arrays `r` and `z` move between processor-1 and other processors. We can eliminate this unnecessary movement using code transformations for data segregation. In this case, we can parallelize the `icdot` calls using OpenMP. This allocates segments of `r` and `z` arrays to specific processors thereby eliminating unnecessary data movement. More significantly, `icdot` calls now operate in parallel. This potentially has a much bigger impact on performance than the elimination of data movement alone.

Similar transformations are carried out for other references from Group-1, which we do not further discuss here.

Group 2: In-region false sharing invalidations constitute almost the entire volume of invalidations for these references. The number of coherence misses closely matches the number of invalidations received. All these references are related to timer routines used for performance benchmarking. Most of the coherence misses arise due to parallel updates to counter arrays indexed by thread id. Since array elements are contiguous, this leads to false-sharing, causing a ping-pong exchange of cache lines across processors. We use intra data-structure padding to align individual array elements at cache line boundaries, which eliminates coherence misses.

Group 3: This group has a single reference exhibiting large volumes of true in-region invalidations. These invalidations occur inside a `omp critical` region updating a shared global clock variable. We eliminate this sharing by maintaining clock variables for each thread separately.

Results: The coherence misses for each reference after optimization are shown in the last column of Figure 7. We see that coherence misses for Groups 2 and 3 have been eliminated (by padding and sharing elimination, respectively) and have decreased significantly for Group 1. Figure 10 shows the wall-clock execution times for the different optimization strategies on the recommended OpenMP data set(`zrad.008.seq`). The readings were obtained on a non-interactive node with 8 OpenMP threads. `DSeg` represents code transformations for data segregation (Group 1 references). `DSeg+Crit` additionally removes the shared global clock (Group 3 reference). `DSeg+Crit+Padding` represents the fully optimized benchmark. We observe that `DSeg` causes significant decrease in wall clock execution time (over 30%), compared to the original program. The performance impact is due to a combination of 2 factors. First, there is reduction in coherence traffic due to our optimizations. Second, the reduction in coherence traffic was achieved by additional parallelization of serial sections of code. This additional speedup also contributes to the overall wallclock time improvement.

It would be hard to achieve these optimizations by conventional time-based profiling alone. Such schemes might be able to pin-point the source-code locations taking significant amounts of execution time. However, our ability to understand the exact *flow* of shared data across processor caches was critical in identifying the ping-pong effect due to insufficient parallelization.

C. SMG2000: Semi-coarsening Grid Solver

SMG2000 is part of the ASCI Purple benchmark set [17]. The SMG code utilizes the `hypre` library [22], that can

TABLE VI
SMG: PER-REFERENCE STATISTICS (PROCESSOR-1)

No.	Reference	Group	Coherence	Invalidations				Optimization	Optimized
				True		False			Coherence
			Misses	In	Across	In	Across	Strategy	Misses
1	rp[]_Read	**1**	170046	0	0	**156585**	13387	Code Transforms	**256**
2	rp[]_Read		83509	0	0	**80145**	3529	for coarse-level	**0**
3	rp[]_Write		43640	0	0	**43305**	3373	interleaving	**0**
4	xp[]_Write		23193	0	0	**22309**	1284		**2764**
5	num_threads	**2**	44362	**44929**	0	0	0	Remove sharing	**0**

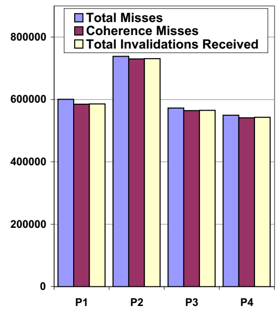


Fig. 11. Breakdown of L2 misses

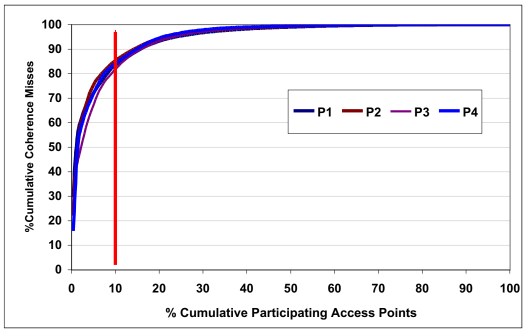


Fig. 12. SMG: Cumulative L2 Coherence Misses

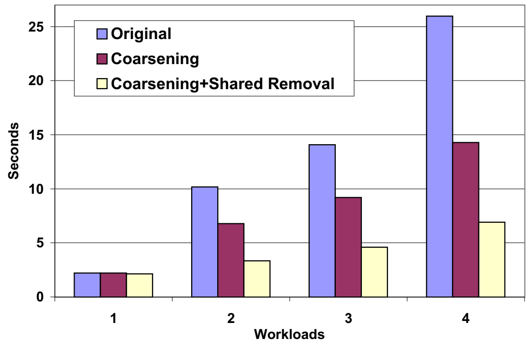


Fig. 13. SMG: Time for different Workloads

select between OpenMP and MPI parallelization. We use the default settings of SMG2000 for our analysis (`10 x 10 x 10 grid, cx=cy=cx=1.0`). We then compare the wall-clock execution time for the recommended full-sized workloads for different optimization strategies.

Analysis: For all processors, the L2 miss rate is quite high, ranging from 64% to 81%. Figure 11 shows that almost all of the L2 misses are coherence misses. It also shows that the number of invalidations received is very close to the number of coherence misses. This indicates that almost all invalidations received by the L2 cache eventually caused a coherence miss.

Our instrumentation framework instrumented 11,047 memory access points, out of which only 338 access points (3%) experienced coherence misses. Figure 12 shows the cumulative coherence misses for the access points that experienced coherence misses ("participating access points") for each processor. Notice that the cumulative distribution is quite skewed — 10% of the participating access points accounted for 82-85% of the total coherence misses for a processor. Thus, by focusing on optimizing the coherence misses for the top references, we can remove a large number of coherence misses, potentially resulting in a significant performance gain.

The per-reference statistics for the top 5 references from processor-1 are shown in Table VI. The statistics for other processors were similar to those of processor-1. As with IRS, we classify references into groups based on coherence characteristics to facilitate analysis.

Group 1: References in this group are all array access references. All references experience a very large volume of in-region false-sharing invalidations. This indicates that multiple processors are updating different data elements on the same cache line, causing the cache line to ping-pong between L2 caches of different processors. The cause of the large volume of invalidations lies in the sub-optimal implementation of loop-level parallelization by the `hypre` library function. This function must choose one loop of a triply-nested loop nest to parallelize. Each loop in the nest iterates over a single coordinate axis. The order of iteration is x,y,z from the inner to the outer loop. The function always chooses the largest dimension for parallelization, with the default being the innermost loop (x dimension). This results in fine-grained interleaving of thread accesses to adjacent array elements, resulting in large amounts of coherence traffic. To correct this, we hoist the OpenMP parallelization to the outermost loop (z dimension) ensuring that threads access data on different cache lines.

Group 2: This group has a single store reference that exhibits large volumes of true-sharing in-region invalidations. The data element referenced is a shared variable that is simultaneously updated by all threads with the number of runnable OpenMP threads, inside an `omp parallel` construct. We eliminate this sharing by replacing the `omp parallel` construct with separate calls to `omp_get_max_threads()` in each thread.

Results: The coherence misses after optimization are shown in the last column of Table VI. Our optimizations have eliminated almost all the coherence misses for these references. We compare the performance impact of our optimizations on wall clock execution time for the following workloads, as recommended by the SMG2000 benchmarking criteria:

```
1. 35x35x35 grid, OpenMP threads=1
2. 35x35x70 grid, OpenMP threads=2
3. 35x70x70 grid, OpenMP threads=4
4. 70x70x70 grid, OpenMP threads=8
```

All workloads have processor configuration 1x1x1 (`-P 1 1 1`), cx=0.1, cy=1.0, cz=10.0. The workloads scale up the input grid size with increasing number of threads keeping the overall data processed per processor constant. Figure 13 compares the wall-clock times for the different workloads. says: 2293500 `Coarsening` represents code transformations for coarse-level interleaving of accesses (Group 1). `Coarsening+Sharing Removal` additionally removes unnecessary shared data access (Group 2). We observe that both optimizations have significant impact on execution time, with a maximum improvement of 73% for the 4th workload (8 OpenMP threads).

D. AMMP: Molecular Mechanics Program

AMMP is a part of the SPEC2001M OpenMP benchmark suite [18]. We use the smaller `test` data set for characterizing the coherence behavior of the benchmark, and later use the larger `train` data set for measuring the performance improvements on the target machine. The benchmark was run with 4 OpenMP threads. We modified the scheduling policy specified by the program to `static` scheduling, from `guided` scheduling, for more repeatable performance numbers. [5] As before, we bound the OpenMP threads to separate processors using the `bindprocessor` system call.

For the coherence characterization, the address traces were obtained on a 8-way SMP Power4-II platform[6]. We updated the coherence simulator configuration to simulate the cache configuration of this target platform, including shared L2 caches. We simulate the generic MESI protocol and do not model the more specialized version of the protocol as implemented on the target POWER4 platform. Table VII shows the top references exhibiting coherence misses for processor 3. The results for other processors are similar.

Invalidator Lists: Figure 14 shows the invalidator lists for selected references. We shall describe invalidator lists in more detail, since this is the first use case to use this feature. The invalidator lists are shown graphically in the following format. Each ellipse represents a reference in the source code. An edge from ellipse `A` (source) to `B` (target) denotes that `A` caused the memory line resident in some other processor's cache to be invalidated, and that memory line was previously accessed by the reference `B`. Here, `A` must be a store reference (since it caused an invalidation) and `B` can be either a load or a store reference. The numbers on the edges denote the percentage of the invalidations of the target reference that were accounted for by the source reference. *E.g.*, consider the invalidator list for reference Ref7 in Figure 14. Ref7 is `a_number_Read`, with source code location atoms.c:111. The data brought into the cache by this reference was invalidated 50% of the time by reference `highest_Write` (atoms.c::235) executing on processor1, 25% of the time by reference `last_Write` (atoms.c::207) executing on processor1 and 25% of the time by reference `last_Write` executing on processor2. The invalidator references are accessing a different data element than the reference being invalidated (`highest_Write`, `last_Write` *vs.* `a_number_Read`). The invalidations occur because all these data elements are resident on the same cache line (an example of *false-sharing*).

Analysis: We have grouped references showing similar characteristics. Let us consider each group in more detail.

Group 1: There are 3 references in this group. Together, they account for 72% of all the L2 coherence misses suffered by this processor. For this group, almost all the invalidations received are in-region true-sharing invalidations, *i.e.*, other processors wrote to the same shared data element within the same OpenMP region causing the invalidation.

The invalidator lists for reference `(a2->qzz)_Read` are shown in Figure 14. It is apparent that all the invalidations for this reference occur due to writes by processors 1 and 2 on the same source code line. In turn, these references are invalidated by the same write instruction on processors 3 and 4. The cycle of invalidations causes a ping-pong exchange of data across the processor caches.

A look at the source code shows why the ping-pong exchange is occurring. All the references access nodes of type `struct atom`. Consider reference `a2->qzz)_Read` at rectmm.c::1237.

```
1158:for( i=0; i< nng0; i++)  {
1160: a2 = (*atomall)[natoms*o+i];
1180: omp_set_lock(&(a2->lock));     //capture lock
1237:   a2->qzz -= (k2*(zt2 - third) + ...);
1306:   omp_unset_lock(&(a2->lock));//release lock
1309: }//end loop
```

For each atom in the for loop, the shared atom data is accessed in a critical section guarded by the `a2->lock` OpenMP lock variable. Our results indicated that the update of the `a2->qzz` element suffers frequent coherence misses due to writes to the same element by different processors.

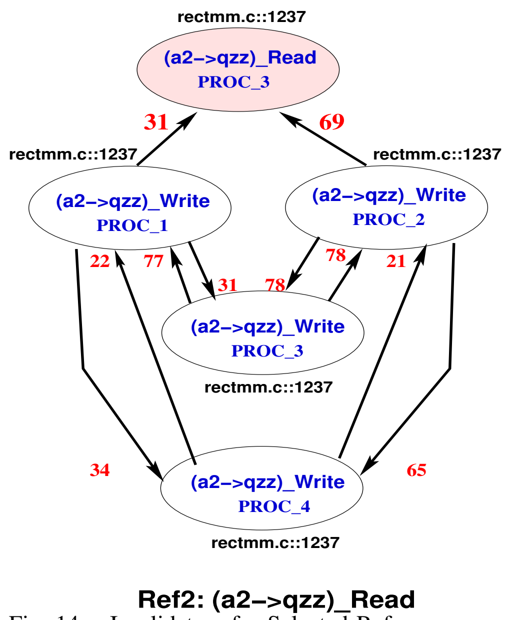
It is difficult to re-structure the code to remove sharing of the atom elements. Instead, we use *prefetching* to preload the data that will be accessed in the near future by this processor using the POWER4 `dcbt` ("Data Cache Block Touch") instruction. Prefetching is beneficial even with larger data sets when the working set size increases beyond the L2 cache capacity and most of the data is fetched from memory

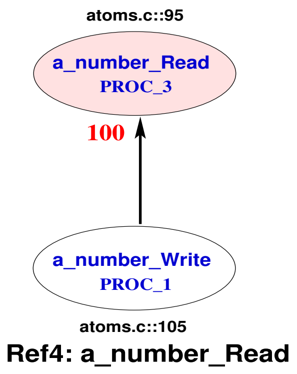
[5]Static scheduling ensures that the each processor executes the same iterations, over multiple runs of the program. With guided scheduling, the iterations that are executed on a processor can vary across multiple program run, leading to more variance in performance numbers.

[6]The Power3 machine that we used for earlier experiments was no longer in service.

TABLE VII
AMMP: PER-REFERENCE STATISTICS

#	Reference			Group	Coh.	Invalidations				Optimization
						True		False		
	file	line	name		misses	in	across	in	across	strategy
1	rectmm.c	1184	(a2->py)_Read		45321	**88081**	0	4489	0	
2	rectmm.c	1237	(a2->qzz)_Read	1	43905	**89614**	0	0	0	**Prefetch**
3	vnonbon.c	536	(a2->dy)_Read		6764	**35700**	1639	0	0	
4	atoms.c	95	a_number_Read		9580	0	**9582**	0	0	
5	atoms.c	99	new_Write		2395	0	**2395**	0	0	**Remove**
6	atoms.c	194	(*name)_Read		2394	0	**2395**	0	0	**Superfluous**
7	atoms.c	111	a_number_Read	2	9582	0	1	**9581**	0	**Parallelization**
8	atoms.c	144	a_number_Read		4791	0	0	**4792**	0	
9	atoms.c	115	serial_Read		2394	0	0	0	**2395**	
10	atoms.c	115	serial_arr[ ]_Write		2095	0	0	0	**2395**	
11	atoms.c	116	serial_p[ ]_Write		2095	0	0	0	**2395**	





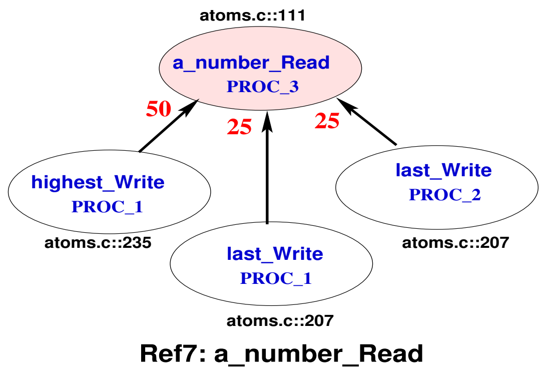


Fig. 14. Invalidators for Selected References

rather than from another processor's L2 cache. We apply this optimization for all the 3 references in this group. The resulting performance improvements are discussed below.

Group 2: All references in group 2 belong to the function `atom()` in atom.c. There are 3 distinct reference sub-groups receiving true-inregion, false-inregion and false-across-region invalidations, respectively. Figure 14 shows the invalidators for reference4 `a_number_Read` (atoms.c::95) and reference7 `a_number_Read` (atoms.c::111). Reference-4 is invalidated always by a write in processor-1 occurring at atoms.c::105. Reference7 suffers false-sharing invalidations due to writes to the shared variables `highest` and `last` in other processors.

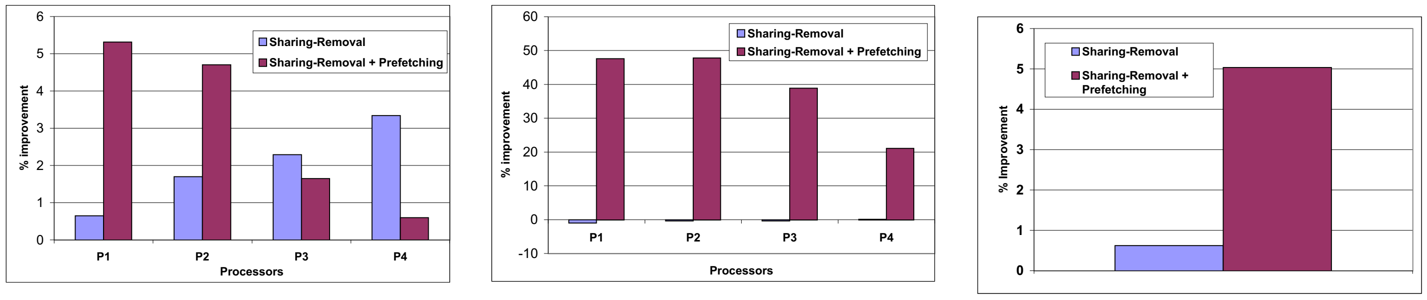
We further reduce the coherence misses for this group as follows. Consider reference4 (atoms.c::95) and its invalidator (atoms.c::105) in the source code.

```
94:#pragma omp parallel
95: if (omp_get_thread_num() ==
        a_number % omp_get_num_threads())
96:   {
97:     new=malloc(ALONG);
98:   }
        .......
105: a_number++;
```

Variable a_number increases linearly with each call to the atom() function. The "if" condition is only satisfied by one thread for each call, so the parallel region is extremely imbalanced. Following the OpenMP region, a_number is updated by the master thread (line 105), which causes a coherence miss on other processors when they attempt to read a_number the next time. We can avoid this needless coherence miss and eliminate the overhead of spawning the parallel region by removing this superfluous parallelization. There are 2 other similar OpenMP regions that are superfluous; they together cause all the other coherence misses in this group. We shall remove OpenMP parallelization for these regions and denote this optimization "Shared-Removal" in the performance results discussed below.

Results: In this section, we compare the performance of the original version of ammp with our optimized versions (Shared-Removal and Shared-Removal + Prefetching). Since our simulator currently does not simulate the effect of prefetch instructions, we do not show the simulator results for the optimized versions. Instead, we measure the performance on the real physical machine using hardware performance counters, shown in Figure 15. For these experiments, we used the larger `train` data set as input. The performance measurements were obtained for each bound OpenMP thread using 4 threads on a non-interactive Power4-II 8-way SMP node. For maximum performance, we force the threads to busy-wait by setting the XLSMPOPTS environment variable to "spins=0:yields=0". The counter values were averaged over 4 runs. We observed very low deviation among runs with a coefficient of variance less than 0.6 for all counter values.

Figure 15(a) and 15(b) shows the reduction in per-processor cycles and per-processor L1 data cache misses, over the original version. Figure 15(c) shows the reduction



(a) Reduction in Processor Cycles Measured with Hardware Counters

(b) Reduction in L1 Data Cache Misses Measured with Hardware Counters

(c) Reduction in Wall-Clock Time

Fig. 15. Reduction in Execution Metrics for AMMP

in wall-clock time for the application. We observe that `Sharing-Removal` leads to a measurable decrease in the number of cycles for each processor and negligible reductions in the overall wall-clock time. This is because the time spent in the atom() function is less significant compared to the overall execution time. The impact of this optimization may increase with a larger number of processors, especially in cc-NUMA systems where the cost of accessing remote memory and remote caches is higher than the cost of accessing their local counterparts [23].

`Sharing-Removal + Prefetching` dramatically decreases the magnitude of L1 data cache misses for all processors, ranging from 21% to 47% across processors. This leads to a 0.5% to 5.3% reduction in processor cycles. Overall, `Sharing-Removal+Prefetching` leads to a 5% reduction in wall-clock time.

E. Other benchmarks

In addition to the benchmarks discussed above, our framework was able to find incorrect/sub-optimal instances of parallelization in several other benchmarks — sPPM from the ASCI Purple suite [17], 301.wupwise_m from the SPEC OMP2001M suite and FT from the C OpenMP version of the NAS-2.3 suite [24]. We discuss them briefly below.

sPPM/ASCI-Purple: Our framework pin-pointed a large number of in-region true-sharing invalidations that were not protected by locks (initbuf() function in sppm/main.m4). The code is shown in Figure 16. The PLOOP macro is expanded by the m4 preprocessor to OpenMP pragmas. Due to incorrect parallelization, all threads update the `mm1, mm2, mm3, mm4, mm5` scalar variables that are used in the body of the loop without critical sections. This is reflected in our coherence simulation results as true-sharing in-region invalidations. However, program correctness is not affected because the values of the overwritten variables are monotonically increasing and are used as indices for initializing array elements to 0. Thus, some array elements may be initialized multiple times, but the problem does not affect program correctness. Also, the initialization only happens once and does contribute significantly to the overall execution time. This problem manifests due to a combination of incorrect parallelization and multiple updates spread over 50 lines of code. It would be very hard to detect this problem by mere visual inspection.

310.wupwise_m/SPEC-OMP2001M: Our framework found two instances of sub-optimal parallelization (rndcnf() and rndphi() functions). The concerned code for rndcnf() is shown in Figure 17. The `U` array is initialized to 0 in parallel, but it is immediately overwritten by the serial thread in the following do loop. This shows up in our simulation results as large across-region true-sharing invalidations by thread 0 (master thread). A similar situation arises in the rndphi() function. The initialization to 0 can be removed. Furthermore, the second DO loop may be parallelized. However, these two functions do not contribute significantly to the overall execution time.

FT/NAS-2.3-C: Our framework found large numbers of in-region false-sharing invalidations and coherence misses in the loop nest shown in Figure 18 (function compute_indexmap() of ft.c). All the invalidations and coherence misses occurred for the update of the `indexmap` variable on line 436. A closer inspection of the loop nest shows the problem: The `i` loop is parallelized but the `i` variable indexes the contiguous dimension of the array `indexmap`. As a result, multiple threads write simultaneously to adjacent elements of `indexmap` located in the same cache line, which leads to a ping-pong exchange of the memory line between processors. This problem is similar to the "coarsening" problem discussed for SMG2000 (Section IV-C). The problem can be alleviated by reordering the loop nest in memory order (k,j,i) and parallelizing the k loop instead. We found significant improvement in execution time for the loop nest after this optimization. However, the compute_indexmap() function is not invoked after the initialization phase. Hence, the optimization had negligible impact on the overall program execution time.

V. Related Work

There are several software-based and hardware-based approaches for memory performance characterization of shared memory multiprocessor systems. Gibson *et al.* provides a good overview of the trade-offs of each approach [25]. At one end of the spectrum are complete software machine

```
1042:  PLOOP(ii,1,iq,11,<<
          ........
          do jj=1,iq*ndata*nbdy
             mi_xma(mm1+jj) = zero
             ........
          enddo
1054:  mm1 = mm1 + iq*ndata*nbdy
          ........
1061:  mm2 = mm2 + iqb*ndata*nbdy
          ........
1072:  mm3 = mm3 + (nbdy*2+iq)*ndata*nbdy2
          ........
1083:  mm4 = mm4 + ndata*2*nbdy*nbdy2
          ........
1090:  mm5 = mm5 + ndata*2*nbdy2*nbdy2
1092:  >>)
```

Fig. 16. sPPM, initbuf() in main.m4

```
48: !$OMP PARALLEL DO
49: DO I=1,LENGTH
50:       U(I) = 0.0
51: ENDDO
52: DO 100 I=1,LENGTH
53:       U(I) = DLARND(2,SEED)
54: 100 CONTINUE
```

Fig. 17. 310.wupwise_m, rndcnf() in rndcnf.f

```
427: #pragma omp for
428: for (i = 0; i < dims[2][0]; i++){
429:      ii = (i+1+xstart[2]-2+NX/2)%NX - NX/2;
430:      ii2 = ii*ii;
431:      for (j = 0; j < dims[2][1]; j++) {
432:           jj = (j+1+ystart[2]-2+NY/2)%NY - NY/2;
433:           ij2 = jj*jj+ii2;
434:           for (k = 0; k < dims[2][2]; k++) {
435:                kk = (k+1+zstart[2]-2+NZ/2)%NZ - NZ/2;
436:                indexmap[k][j][i] = kk*kk+ij2;
437:           }
438:      }
439: }
```

Fig. 18. FT, compute_indexmap in ft.c

simulators. RSim is a simulator for ILP multiprocessors with support for CC-NUMA architectures with a invalidation-based directory mapped coherence protocol [4]. SimOS is a complete machine simulator capable of booting commercial operating systems [6]. However, these frameworks simulate hardware and architecture state to a great detail, increasing simulation overhead. This limits the size of the programs and workloads that they can run. In contrast, ccSIM is an event-based simulator that simulates only memory hierarchies. Our instrumentation tool is flexible and allows us to collect partial traces of only the pertinent memory access. Thus, we can handle a much larger range of programs and workloads. More importantly, these simulators provide only bulk statistics intended for evaluating architecture mechanisms. In contrast, we aim at providing the application programmer with information on the shared-memory behavior of the program and correlate metrics to higher levels of abstraction, such as line numbers and source code data structures.

Execution-driven simulators are a popular approach for implementing memory access simulators. Code annotation tools annotate memory access points. Annotations call handlers, which invoke the memory access simulator. Augmint [5], Proteus [1] and Tango [3] are examples of this approach. All these tools use static code annotation, *i.e.*, they annotate the target code at the source, assembly or object code level. MemSpy [26] and CProf [27] are cache profilers that aim at detecting memory bottlenecks. CProf relies on post link-time binary editing through EEL [28], [29]. Lebeck and Wood also applied binary editing to substitute instructions that reference data in memory with function calls to simulate caches on-the-fly [30]. Other approaches rely on hardware support, such as watchdogs [31] or statistical sampling with hardware support in ProfileMe [32], to gather information on data references. Scal-Tool detects and quantifies scalability bottlenecks in distributed shared memory architectures, such as the SGI Origin 2000 [33]. It determines inefficiencies due to cache capacity constraints, load imbalance and synchronization. Nikolopoulos *et al.* discuss OpenMP optimizations for irregular codes based on memory reference tracing to indicate when page migration and loop redistribution is beneficial. This results in comparable performance of optimized OpenMP with MPI parallelization, again on the Origin 2000 [34].

CProf and MemSpy use static binary rewriting, but they only provide information about uniprocessor misses (cold, capacity, conflict). In contrast, we focus on characterizing shared memory traffic.

All other tools (besides CProf and MemSpy) discussed above do not allow misses to be related to source code and data structures. Furthermore, our work differs from these works in the fundamental approach of rewriting binaries, which is neither restricted to a special compiler or programming language, nor does it preclude the analysis of library routines.

In addition, execution-driven simulators are often tied to one architecture due to the requirements of annotating the code at assembly or object level. DynInst is available on a number of architectures. Porting our framework to these platforms only involves changing the memory instructions to be instrumented. Another major difference addresses the overhead of large data traces inherent to all these approaches. We allow the analysis of partial traces and employ trace compression to provide compact representations.

SM-prof is an aggregate classification tool for shared-memory references resulting in coherence traffic [35]. It classifies *all accesses* into access classes depending on how many processors read/write to the same data block in the current time slot. It is up to the analyst to find and quantify the location and magnitude of the coherence bottleneck. The analysis tool does not provide this information at the level of individual access points, but only at the level of each access class. This causes to authors "to suspect false sharing" [35]. In contrast, ccSIM is a per-reference coherence analysis tool. We generate detailed coherence statistics for *each access* point, as well as for global data structures. Metrics include the magnitude of coherence misses, true and false sharing invalidations and classification of invalidations across and in a parallel OpenMP regions. Thus, we do not suspect, we *know* when false/true sharing occurs (among other symptoms).

The SIGMA (Simulator Infrastructure to Guide Memory Analysis) [36] system has many similarities with our work. It uses post-link binary instrumentation and online trace compression, and allows tagging of metrics to source code constructs. A toolkit by Marin and Mellor-Crummey uses statistical methods based on dynamic measurements of edge counters and histograms of reuse distances for each memory reference to predict cache and execution behavior across different architectural platforms [37]. Both of these approaches are limited to uniprocessor systems while we focus on analyzing coherence traffic for SMPs. The latter work does not focus on transformations, unlike our work.

Recently, most architectures have added hardware counters that provide information on the frequency of hardware events, *e.g.*, to count shared memory events. Portable APIs like PAPI provide a reasonably platform-independent method of accessing these counters [38]. Hardware counters impose no runtime overhead, and querying counters is typically of low overhead. However, they only provide aggregate statistics without any relation to the source code, and there are only a limited number of counters available. In addition, there are often restrictions on the type of events that can be counted simultaneously. HPCToolkit uses statistical sampling of performance counter data and allows information to be correlated to the program source [39]. Our method goes beyond this granularity by identifying evictors within caches and coherence traffic in SMP to indicate source of inefficiency. A number of commercial tools, such as Intel's VTune, SGI's Speedshop, Sun's Workshop tools also use statistical sampling with source correlation, albeit at a coarser level that HPCToolkit or our approach. It is possible to finer-grained information with customized hardware. The FlashPoint system uses a custom system node controller to monitor coherence events [25]. In general, hardware monitors are fast but may constrain the number of events that can be monitored. At this point in time, they lack a wide acceptance in practice.

Krishnamurthy and Yelick develop compiler analysis and optimization techniques for the shared-memory programming paradigm using SplitC as an example [7]. Their main concern is the hardware-supported coherence model, namely weak consistency. They are specifically concerned about writes and invalidations occurring out-of-order. Their optimizations reflect the constraints of reordering writes in the presence of locks and barriers with respect to weak consistency and employ message pipelining (aggregation of writes) and reduction of communication (two-way to one way or elimination). Satoh *et al.* study compiler optimizations for OpenMP in a distributed shared memory system based on data-flow techniques to analyze thread interactions [8]. Optimizations include barrier removal and data privatization to reduce coherence-induced messages. Our work shares the aim at optimizing shared-memory applications with these approaches. However, we take a radically different approach by analyzing traces to determine if and where inefficiencies in terms of coherence traffic exist and if there is room for improvements.

VI. Conclusion

This work describes a novel framework to analyze cache coherence and to correlate detailed information back to source-code constructs. At the center of our framework is ccSIM, a cache-coherent memory simulator. This simulator obtains coherence metrics and retains reference correlations based on actual data traces. The traces are obtained *via* on-the-fly dynamic binary rewriting of OpenMP benchmarks executing on a contemporary SMP architecture. We explored the degrees of freedom in interleaving data traces from the different processors with respect to simulation accuracy compared to hardware performance counters. We evaluated the run-time overhead of software instrumentation and several on-line trace compression algorithms. We also provided detailed coherence information per data reference and relate them to their data structures and reference locations in the code.

Experimental results indicate a close match between our simulations and the observed hardware performance counters for coherence events. By deriving detailed coherence information, it becomes feasible to indicate the location of invalidations in the application code. Benefits of this detailed level of information are demonstrated by our ability to infer opportunities for optimizations. Without ccSIM, these sources of coherence bottlenecks would not have easily been detected and, more importantly, localized. The resulting program transformations ranged from coarsening of access granularity over data alignment to call parallelization, critical section removal with privatization and prefetching. Measurements of optimized codes showed both significantly decreased coherence traffic and execution time savings.

References

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

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

[3] H. Davis, S. R. Goldschmidt, and J. Hennessy, "Multiprocessor simulation and tracing using tango," in *Proceedings of the 1991 International Conference on Parallel Processing*, vol. II, Software. Boca Raton, FL: CRC Press, Aug. 1991, pp. II–99–II–107.

[4] C. Hughes, V. Pai, P. Ranganathan, and S. Adve, "Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors," *IEEE Computer*, vol. 35, no. 2, pp. 40–49, February 2002.

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

[6] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, "Complete computer system simulation: The SimOS approach," *IEEE parallel and distributed technology: systems and applications*, vol. 3, no. 4, pp. 34–43, Winter 1995. [Online]. Available: http://www.computer.org/concurrency/pd1995/p4034abs.htm; http://dlib.computer.org/pd/books/pd1995/pdf/h40034.pdf

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

[8] S. Satoh, K. Kusano, and M. Sato, "Compiler optimization techniques for openMP programs," *Scientific Programming*, vol. 9, no. 2-3, pp. 131–142, 2001.

[9] J. Vetter and F. Mueller, "Communication characteristics of large-scale scientific applications for contemporary cluster architectures," in *International Parallel and Distributed Processing Symposium*, Apr. 2002.

[10] ——, "Communication characteristics of large-scale scientific applications for contemporary cluster architectures," *Journal of Parallel Distributed Computing*, vol. 63, no. 9, pp. 853–865, Sept. 2003.

[11] B. Buck and J. K. Hollingsworth, "An API for runtime code patching," *The International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, Winter 2000.

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

[13] *Official OpenMP Specification*, www.openmp.org, May 2005. [Online]. Available: http://www.openmp.org/drupal/mp-documents/spec25.pdf

[14] J. Marathe, A. Nagarajan, and F. Mueller, "Detailed cache coherence characterization for openmp benchmarks," in *International Conference on Supercomputing*, June 2004.

[15] H. Jin, M. Frumkin, and J. Yan, "The openmp implementations of nas parallel benchmarks and its performance," NASA Ames Research Center, TR NAS-99-011, Oct. 1999.

[16] W. Gunsteren and H. Berendsen, "Gromos: Groningen molecular simulation software," Laboratory of Physical Chemistry, University of Groningen," TR, 1988.

[17] LLNL, "Asci purple codes," 2002, http://www.llnl.gov/asci/purple.

[18] SPEC, "SPEC OMPM2001 benchmarks," 2001, http://www.spec.org/omp.

[19] J. Marathe, "METRIC: Tracking memory bottlenecks via binary rewriting," Master's thesis, North Carolina State University, June 2003.

[20] M. F. Oberhumer, "LZO real-time data compression library," 2002. [Online]. Available: http://www.oberhumer.com/opensource/lzo/

[21] J. Marathe, F. Mueller, and B. de Supinski, "A hybrid hardware/software approach to efficiently determine cache coherence bottlenecks," in *International Conference on Supercomputing*, June 2005, pp. 21–30.

[22] R. D. F. E. Chow, A. J. Cleary, "Design of the hypre preconditioner library," in *SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing*, Oct. 1998.

[23] J. Marathe and F. Mueller, "Hardware profile-guided automatic page placement for ccnuma systems," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mar. 2006, pp. 90–99.

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

[25] J. Gibson, "Memory profiling on shared memory multiprocessors," Ph.D. dissertation, Stanford University, July 2003.

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

[27] A. R. Lebeck and D. A. Wood, "Cache profiling and the SPEC benchmarks: A case study," *Computer*, vol. 27, no. 10, pp. 15–26, Oct. 1994.

[28] J. Larus and T. Ball, "Rewriting executable files to measure program behavior," *Software Practice & Experience*, vol. 24, no. 2, pp. 197–218, Feb. 1994.

[29] J. R. Larus and E. Schnarr, "EEL: Machine-independent executable editing," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1995, pp. 291–300.

[30] A. R. Lebeck and D. A. Wood, "Active memory: A new abstraction for memory system simulation," *ACM Transactions on Modeling and Computer Simulation*, vol. 7, no. 1, pp. 42–77, Jan. 1997.

[31] B. R. Buck and J. K. Hollingsworth, "Using hardware performance monitors to isolate memory bottlenecks," in *Supercomputing*, ACM, Ed., 2000, pp. 64–65. [Online]. Available: http://www.sc2000.org/proceedings/techpapr/papers/pap197.pdf

[32] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos, "ProfileMe: Hardware support for instruction-level profiling on out-of-order processors," in *Proc. 30th Annual IEEE/ACM Int. Symp. on Microarchitecture (MICRO-97)*, Dec. 1997, pp. 292–302.

[33] Y. Solihin, V. Lam, and J. Torrellas, "Scal-tool: Pinpointing and quantifying scalability bottlenecks in dsm multiprocessors," in *Supercomputing*, Nov 1999.

[34] D. Nikolopoulos, C. Polychronopoulos, and E. Ayguade, "Scaling irregular parallel codes with minimal programming effort," in *Supercomputing*, 2001.

[35] M. Brorsson, "A tool to visualise and find cache coherence performance bottlenecks in multiprocessor programs," in *ACM SIGMETRICS Conference*, May 1995, pp. 178–187.

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

[37] G. Marin and J. Mellor-Crummey, "Cross architecture performance predictions for scientific applications using parameterized models," in *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2004, p. (to appear).

[38] S. Browne, J. Dongarra, N. Garner, K. London, , and P. Mucci, "A scalable cross-platform infrastructure for application performance tuning using hardware counters," in *Supercomputing*, Nov. 2000.

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

Frank Mueller (mueller@cs.ncsu.edu) is an Associate Professor in Computer Science and a member of the Centers for Embedded Systems Research (CESR) and High Performance Simulations (CHiPS) at North Carolina State University. Previously, he held positions at Lawrence Livermore National Laboratory and Humboldt University Berlin, Germany. He received his Ph.D. from Florida State University in 1994. He has published papers in embedded and real-time systems, compilers and parallel/distributed systems. He is a member of ACM SIGPLAN & SIGBED, the IEEE Computer Society and a Senior Member of the ACM and the IEEE. He is a recipient of an NSF Career Award, an IBM Faculty Award and a Fellowship from the Humboldt Foundation.

Jaydeep Marathe (jp-marath@ncsu.edu) is a doctoral student in the Computer Science Department of the North Carolina State University. He is interested in parallel performance evaluation and optimization.