Abstract

MARATHE, JAYDEEP P. Trace Based Performance Characterization and Optimization. (Under the direction of Associate Professor Frank Mueller).

Processor speeds have increased dramatically in the recent past, but improvement in memory access latencies has not kept pace. As a result, programs that do not make efficient use of the processor caches tend to become increasing *memory-bound* and do not experience speedups with increasing processor frequency.

In this thesis, we present tools to characterize and optimize the memory access patterns of software programs. Our tools use the program's *memory access trace* as a primary input for analysis. Our efforts encompass two broad areas — performance analysis and performance optimization. With performance analysis, our focus is on automating the analysis process as far as possible and on presenting the user with a rich set of metrics, both for single-threaded and multi-threaded programs. With performance optimization, we go one step further and perform automatic transformations based on observed program behavior.

We make the following contributions in this thesis. First, we explore different tracing strategies — software tracing with dynamic binary instrumentation, hardware-based tracing exploiting support found in contemporary microprocessors and a hybrid scheme that leverages hardware support with certain software modifications. Second, we present a range of performance analysis and optimization tools based on these trace inputs and additional auxiliary instrumentation. Our first tool, METRIC, characterizes the memory performance of single-threaded programs. Our second tool, ccSIM extends METRIC to characterize the coherence behavior of multithreaded OpenMP benchmarks. Our third tool extends ccSIM to work with hardware-generated and hybrid trace inputs. These three tools represent our performance analysis efforts. We also explore automated performance optimization with our remaining tools. Our fourth tool uses hardware-generated traces for automatic page placement in cache coherent non-uniform memory architectures (ccNUMA). Finally, our fifth tool explores a novel trace-driven instruction-level software data prefetching strategy.

Overall, we demonstrate that memory traces represent a rich source of information about a program's behavior and can be effectively used for a wide range of performance analysis and optimization strategies.

Trace Based Performance Characterization and Optimization

by

Jaydeep Marathe

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh,NC

2007

Approved By:

Dr. Yan Solihin	Dr. Tao Xie
Dr. Frank Mueller Chair of Advisory Committee	Dr. Vincent Freeh

Dedication

To my parents. Without their constant love, support and guidance, this thesis would not have been possible.

Biography

Jaydeep Marathe grew up in Pune, India. He graduated with a Bachelors Degree in Computer Engineering from Pune University in 2000. He received a Masters degree in Computer Science from North Carolina State University in 2003. With the defense of this dissertation, he will receive a PhD in Computer Science from North Carolina State University, in July 2007. Out in the real world, he will look back and have fond memories of his time as a student researcher.

Acknowledgements

First and foremost, I would like to thank Dr. Frank Mueller, my graduate adviser. He has been great to work with, and always allowed me to explore many different research ideas. His door was always open for discussion. I learnt a lot from him about doing research as well as writing and presentation. I would also like to thank Dr. Freeh, Dr. Xie and Dr. Solihin for serving on my thesis committee. Finally, I am grateful to Dr. Mueller and the Computer Science Department of North Carolina State University for financial assistance and for giving me the opportunity for graduate studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Processor speeds have increased dramatically in the recent past. However, the rate of improvement in DRAM access latencies has been slower, as shown by Figure 1.1. The increasing difference between the performance of the CPU and the memory system has serious implications for contemporary software programs. Another issue is the advent of multicore platforms that are being introduced by all major processor vendors. As the number of cores increase, the demand for memory bandwidth grows, even without increasing the individual processor frequency.

As a result, programs that do not make efficient use of the processor cache will tend to become increasingly *memory bound* and will not experience significant speedups with increasing CPU frequency. In fact, programs may even experience performance degradation as the number of cores increases without corresponding increase in memory bandwidth.

In our work we address this "memory wall"[112] in two broad directions. First, we build automated tools that help the programmers to characterize and *understand* the memory access patterns of their programs. Our focus is on automating the performance analysis process as far as possible and on presenting the user with a rich set of metrics, both for single-threaded and multi-threaded programs. Second, we go one step further and perform automatic performance *optimization* based on observed program behavior.

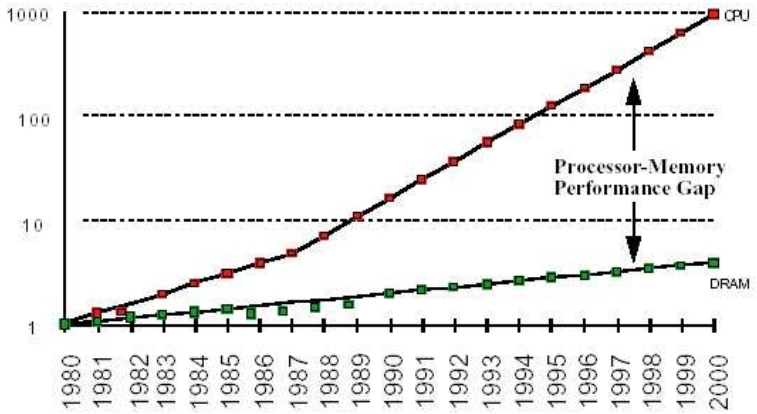


Figure 1.1: Improvement in CPU and DRAM speeds [39]

1.1 Problem Statement

How can a program's memory trace be used to analyze its behavior and optimize its performance?

This problem statement has several facets:

1. What are the different methods to obtain the program's memory access trace?
2. What are the *quality* versus *overhead* tradeoffs between these trace generation methods?
3. What are the different performance analysis and optimization frameworks that are feasible using memory traces?
4. What are the *pre-requisites* in terms of trace information and detail that these analyses entail, and is it possible to satisfy them with memory traces and additional instrumentation?

In this thesis, we explore these questions in detail. We demonstrate that memory access traces are a rich source of information about the program's behavior and represent far more profile information that is available with traditional time-based profiling tools like gprof. In the rest of the chapter, we first discuss the types of trace generation that we have developed and used in our tools. Then, we introduce our trace-based analysis and optimization tools.

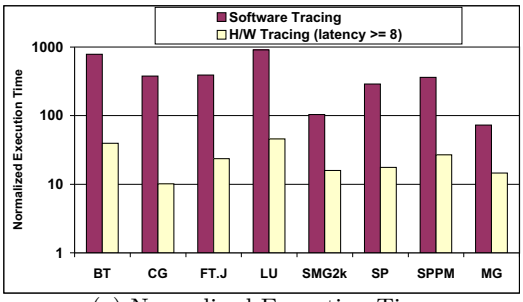
1.2 Trace Generation

In our work, we have used memory traces obtained with both software and hardware means. In the pure software approach, we instrument the memory access instructions of an executing program using dynamic binary instrumentation. Instrumentation at the level of executable instructions enables whole-program trace-based analysis, even for parts of the program for which source code is not available (*e.g.*, third-party libraries). This approach is portable (requires no explicit hardware support) but slows down programs significantly. Also, the sheer volume of loads and stores, even for programs with small data sets, can be an obstacle for some analyses.

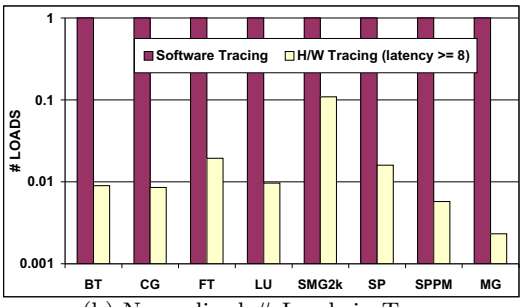
Several processor architectures now have explicit hardware support for extracting memory traces. The Itanium2 and x86 platforms can monitor and capture many load misses. Hardware based tracing has significantly lower overhead (compared to software-based tracing). In addition, the Itanium2 platform allows *filtering* of loads before capture based on the number of cycles the load took to complete. By setting the filter value appropriately, we can ignore loads that hit in the L1/L2/L3 caches. This vastly reduces the number of loads that are captured and thus also the overhead of the tracing process. We show that many analyses can work effectively with such a filtered trace. On the other hand, current hardware implementations have some restrictions on the memory accesses that can be captured. On contemporary Itanium2 and x86 platforms, hardware limitations make the trace capture *lossy*, *i.e.*, only a fraction of the qualified trace elements are actually captured. Also, these implementations have no support for capturing store accesses that are essential for some tools (*e.g.*, coherence traffic analysis).

Figures 1.2(a) and 1.2(b) illustrate the tradeoff between hardware and software tracing. The software-based tracer employs binary instrumentation to obtain the load access stream. The hardware-based tracer exploits the Itanium2's performance monitoring unit to filter out loads taking less than 8 cycles to complete (most L1 and L2 load hits). In addition, the Itanium2 hardware is lossy and drops many of the qualified load samples that should have been captured. We see that the hardware logs about 10 to 100 times *fewer* loads compared to the software tracer, but is more than 10 times faster. In our work, we show that even lossy traces can be effectively used for certain performance analyses and optimizations.

We shall now introduce our trace-based analysis and optimization frameworks



(a) Normalized Execution Time



(b) Normalized # Loads in Trace

Figure 1.2: Comparison of software-based and hardware-based tracing

shown in Figure 1.3. Broadly, our efforts can be divided into automated tools for performance evaluation and performance optimization. Frameworks use either software or hardware generated traces or a combination. A brief introduction of the frameworks follows.

1.3 METRIC: Memory hierarchy analysis for single-threaded benchmarks

The objectives for this work are as follows:

1. To explore the use of memory traces for memory hierarchy performance analysis of single-threaded programs;

2. To explore the use of dynamic binary rewriting for extracting memory traces from an executing program;

3. To test novel light-weight compression strategies for *online* compression of memory traces;

4. To abstract and tag low-level events such as cache misses to high-level source code constructs such as data variables and source code locations, using only symbolic information extracted from the executable;

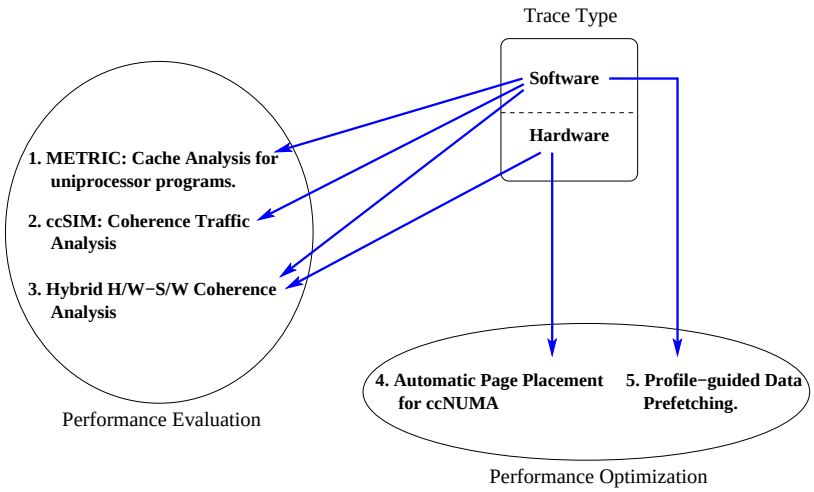


Figure 1.3: Our trace-based frameworks

5. To generate rich metrics that characterize the memory access patterns in the program; and

6. To demonstrate the use of these metrics for effecting novel optimizations that are very hard to achieve with pure static compiler analysis.

We call our tool METRIC ("**ME**mory **TR**acIng without re-**C**ompiling"). METRIC is a trace-based incremental memory hierarchy simulator that generates a rich set of metrics to aid the analysis of single-threaded programs. The trace consists of memory accesses and scope change events (*e.g.,* enter/exit function) and is obtained via software dynamic binary rewriting of executing programs. The trace is compressed online using a novel compression algorithm. The compressed trace is used offline for incremental memory hierarchy simulation and generates a rich set of metrics tagged to high-level source code constructs like data variables and source locations (line::file).

An earlier version of METRIC was presented in my Masters thesis [62]. However, in later work we have significantly changed the compression strategy, generated new results and added unique use cases demonstrating optimizations that are hard to achieve with static compiler analysis. The details of the our new contributions are elaborated in Section 2.2.

1.4 ccSIM: Source-Code Correlated Cache Coherence Characterization of OpenMP Benchmarks

The recent trend towards multi-core and multi-processor shared-memory machines has prompted a shift towards parallel programming. Understanding the sharing of data among threads is essential to detecting and alleviating performance bottlenecks in parallel programs. We explored the use of trace-based analysis for this purpose. The objectives for this work are as follows:

1. To extract traces from multi-threaded OpenMP benchmarks using dynamic binary rewriting, which has not been done before, to the best of our knowledge;

2. To extract *synchronization* information among threads, which is essential for correct performance simulation — this entails *reverse engineering* compiler generated functions for OpenMP directives which is unprecedented in past work;

3. To exploit the per-thread memory traces and OpenMP synchronization information for incremental coherence protocol simulation. A unique property of our approach is the ability to relate low-level coherence events such as invalidations to source code constructions such as line::file tuples. In contrast, past work has considered coherence analysis usually from an *architecture* perspective, instead of the *user* perspective as we do; and

4. To demonstrate the capability of our framework for detecting, understanding and resolving sharing bottlenecks in large real world programs.

We extended METRIC for analysis of coherence traffic of shared memory multithreaded programs that use OpenMP for parallelism. We call our tool ccSIM (**C**ache **C**oherence **SIM**ulator). We demonstrate that our tool can find optimization opportunities in several production codes that have been used for many years on tens or hundreds of processors.

The basic idea and initial prototype of ccSIM were the result of collaborative research with another researcher, Anita Nagarajan. One of the case studies (NBF, Section 3.5.1) is also discussed in her Master's thesis [75]. We include it in this document to provide a comprehensive account of the conducted work.

1.5 Hybrid Hardware/Software Coherence Analysis

The previous tool (ccSIM) uses software instrumentation to extract the memory access trace. In this work, we explore the use of lossy hardware-generated traces for analyzing the coherence traffic at the source code level. The objectives of this work are as follows:

1. To explore the use of the Itanium2 performance monitoring unit (PMU) and high precision timing register to generate filtered memory traces;
2. To measure the execution overhead, trace sizes and degree of lossiness in the PMU-generated trace;
3. To explore software strategies that allow the PMU to trace stores, which it is unable to do natively;
4. To evaluate an alternative *software-centric* approach that uses high precision timing hardware to filter load instructions in the trace; and
5. To compare and contrast both these methods (PMU-centric, software-centric) with respect to overhead, trace sizes and accuracy of coherence simulation. Accuracy is measured with respect to coherence simulation results achieved with full software tracing (as used in our earlier ccSIM work).

We show that both the PMU-centric and software-centric methods can reduce the number of trace accesses that need to be captured by an order of magnitude while retaining the precision of the corresponding coherence simulation for many benchmarks.

1.6 Hardware Profile-guided Automatic Page Placement for ccNUMA

Itanium2 processors are used in popular cache coherence non-uniform memory architecture (ccNUMA) systems that range from 8 to 512 processors. In this work, we explored the use of lossy load traces obtained from the Itanium2 performance monitoring unit (PMU) to achieve profile-guided page placement. Page placement has a large impact

on application performance on ccNUMA machines because remote memory loads are much more expensive compared to loads from local memory. The objectives of this work are as follows:

1. To explore the use of Itanium2 performance monitoring unit to generate traces of filtered long-latency loads for multi-threaded OpenMP programs;
2. To explore the use of these traces for automated page placement without special compiler, linker or operating system support;
3. To support page placement for both static and dynamically allocated regions of memory;
4. To explore the use of an alternative profile source, translation lookaside bufffer (DTLB) misses, for page placement; and
5. To compare and contrast the two profile sources with respect to overhead, quality of the generated page placement and benefits in terms of reduced execution time and remote loads.

1.7 PFetch: Profile-guided Data Prefetching

In contrast to our page placement efforts that tries to *reduce* the average latency of memory access, we also explored trace-based data prefetching that attempts to *hide* it instead. The objectives for this work are as follows:

1. To explore the use of memory traces for a novel software-only data prefetching scheme based on observed cross-instruction address predictability;
2. To *unify* several past approaches that each target a separate source of predictability by introducing a new standard analysis method; and
3. To explore the use of novel threshold-based schemes for addressing the issues of prefetch accuracy, prefetch timeliness and prefetch redundancy.

Our software tracing strategy uses dynamic binary rewriting to extract memory access traces from executables. An advantage of working at the instruction level is that

we can analyze programs across function, module and library boundaries. In this work, we used our software tracing framework to extract memory access traces from single-threaded programs that exhibit significant cache misses. The traces are analyzed to detect instances of *predictability* such that the address of a load miss can be predicted given a certain number of previous memory accesses. This predictability can be leveraged to *prefetch* the missing memory line into cache.

We have introduced our trace-based frameworks and described our objectives for each. In the remainder of the document, we shall describe and evaluate each framework in detail.

Chapter 2

METRIC: Memory Tracing Via Dynamic Binary Rewriting to Identify Cache Inefficiencies

2.1 Summary

With the diverging improvements in CPU speeds and memory access latencies, detecting and removing memory access bottlenecks becomes increasingly important. In this work we present METRIC, a software framework for isolating and understanding such bottlenecks using partial access traces. METRIC extracts access traces from executing programs without special compiler or linker support. We make four primary contributions. First, we present a framework for extracting partial access traces based on dynamic binary rewriting of the executing application. Second, we introduce a novel algorithm for compressing these traces. The algorithm generates constant space representations for regular accesses occurring in nested loop structures. Third, we use these traces for offline incremental memory hierarchy simulation. We extract symbolic information from the application executable and use this to generate detailed source-code correlated statistics including per-

reference metrics, cache evictor information and stream metrics. Finally, we demonstrate how this information can be used to isolate and understand memory access inefficiencies. This illustrates a potential advantage of METRIC over compile-time analysis for sample codes, particularly when interprocedural analysis is required.

2.2 New Contributions

METRIC was first presented in my Master's thesis [62]. However, since then, we have made significant enhancements to the framework and analysis techniques and also generated new results that supersede the data presented originally. In this chapter, we shall present new results from our enchanced framework. We have explicitly cited the Master's thesis for parts of the framework that have not changed significantly in the enhanced version — we retain them here to maintain the readability of the text.

We make the following new contributions in this work, compared to the version reported in the Master's thesis:

- We have re-designed the online compression strategy. The new approach separates compression of trace *ordering* from trace *addresses*. We show that our new strategy achieves better compression than the best known state-of-the-art compression schemes for many of the SPEC FP benchmarks.
- We have added new "stream-oriented metrics" to the pre-existing metrics generated by the analysis component.
- We present 3 new use cases for METRIC that would be very hard or impossible to optimize with static compiler analysis.

The following components of METRIC are mostly unchanged from the Master's thesis version:

- The overall concept of METRIC — using binary instrumentation to extract memory traces, compress them online and use them for offline memory hierarchy simulation.
- The tracing framework based on binary instrumentation.
- The algorithm for trace compression for processing trace addresses (not trace ordering).

- Some metrics generated by the memory hierarchy simulator.

2.3 Introduction

Over the past decade, processor speeds have increased much faster than memory access speeds. Due to this trend, application execution times are increasingly dominated by the time spent in accessing memory. Tools are needed that can efficiently profile the memory access behavior of the program and help in detecting, isolating and understanding *causes* of the potential memory access inefficiencies. In this work we present one such tool, METRIC. METRIC employs incremental memory hierarchy simulation using partial memory access traces and generates detailed high-level metrics characterizing the application's memory use.

Simulation may be performed *offline* using previously extracted access traces or *online* as the application executes. In spite of the accuracy that trace-driven memory simulation affords, efficiency requirements dictate that it be used judiciously. For instance, software tracing incurs high runtime overheads, making full application simulation with reasonable data sets infeasible. Furthermore, even programs with short execution times may generate traces requiring gigabytes of storage. These limitations can be alleviated with *partial* data traces representing a subset of the access footprint of the target. Such traces tend to be comparatively small and less expensive to collect while still capturing the most critical data access points. Our focus is on scientific benchmarks, which generally employ algorithms with convergence criteria that are checked on a regular basis at the end of a *timestep*. The computation of each timestep is highly repetitive and, thus, representative for the overall application behavior, as shown elsewhere [108]. Generating and exploiting partial data traces for online incremental memory hierarchy simulation addresses both high tracing overheads and large storage requirements without sacrificing accuracy. This is the approach we take.

METRIC stands for "MEmory TRacIng without re-Compiling". We draw on previous experience with partial data traces [74] and binary rewriting [63] to detect memory hierarchy bottlenecks. METRIC is also influenced by our work with large scale benchmarks [108], another example of data centric computation where data sizes exceed cache capacities.

In this work we make the following contributions

- We develop an approach that uses dynamic binary rewriting to extract memory access traces from executing applications.
- We develop a novel algorithm for efficient access trace compression of programs with nested loop structures.
- We present a cache analysis methodology (partially based on prior work by Mellor-Crummey *et al.* [70]) that uses partial access traces to generate cache metrics — including detailed evictor information — correlated to high-level constructs such as source code locations and data structures.
- We show how METRIC can be used to understand a diverse range of memory access inefficiencies, some of which are hard to detect with static compiler analysis.

METRIC builds on the DynInst instrumentation framework [10] to exploit *dynamic binary rewriting*, or post link time manipulation of binary executables, enabling program transformation potentially even while the target is executing. Unlike conventional instrumentation, which generally requires compiler interaction (*e.g.*, for profiling) or inclusion of special libraries (*e.g.*, for heap monitoring), this approach obviates requirements of recompiling or relinking.

Dynamic binary rewriting can capture memory references of the entire application, including library routines, and it works equally well for mixed language applications commonly found in production scientific codes [108]. The techniques can be adapted to address changing input dependencies and application modes, *i.e.*, changes over time in application behavior. Furthermore, binary manipulation techniques have been shown to offer new opportunities for program transformations, and these potentially yield performance gains beyond the scope of static code optimization without profile guided feedback [5].

2.4 The METRIC Framework

The METRIC framework, shown in Figure 2.1, uses partial access traces for memory hierarchy simulation. Our framework extracts these comparatively small, low overhead access traces without compiler or linker support, *i.e.*, traces can be extracted from arbitrary

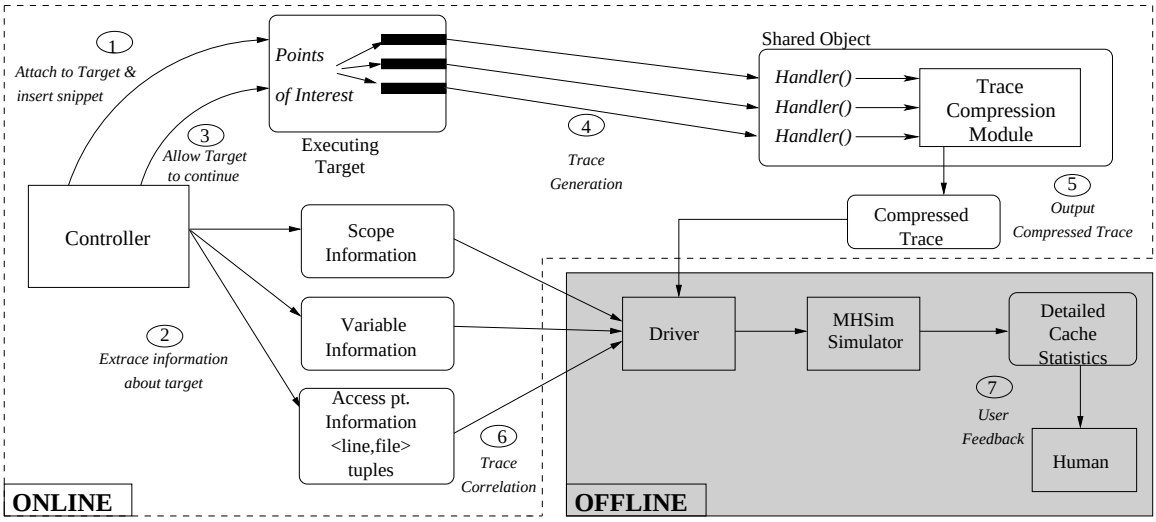


Figure 2.1: The METRIC Framework [62]

executables. To achieve this, we dynamically modify the executing application by injecting instrumentation code via binary rewriting. We instrument memory access instructions to precisely capture the data access stream of the target application, and the user may activate or deactivate tracing so that data reference streams are selectively generated. This facility builds the foundation for capturing partial memory traces.

Figure 2.1 shows two phases in the process of analyzing bottlenecks with METRIC — *online* and *offline*. In the online phase, we instrument the application and extract the memory access trace. After trace generation is complete, the instrumentation is removed and the target application continues its execution without overhead. The traces are then used offline for memory hierarchy simulation in a background process or on a separate processor.

The flow of control is as follows. The user provides the application process id (PID) and the names of the target function(s) to the controller program. The controller program attaches to the executing target and uses DynInst to access the Control Flow Graph (CFG) for these target functions. The text section of the target application is parsed and the *memory access* and *scope change* instructions are instrumented. Scope change instructions transfer control to enter or exit program scopes (such as functions and loop nests). Recording the scope change instructions allows the memory hierarchy simulator to aggregate the generated memory usage metrics at multiple levels of detail (scope) in the

target application's source code. The instrumentation consists of calls to handler functions in a shared library. The shared library is loaded into the target's address space through a special one-shot instrumentation.

Once instrumentation is complete, the target is allowed to continue. As the instrumented application executes, different handler functions in the shared library get invoked, depending on the type of event being recorded, *i.e.*, load, store, enter_scope and exit_scope. The handler functions, in turn, call the compression routines, which attempt to detect regular patterns in the incoming stream. The compression routines maintain statistics about the regularity of the access stream seen at each memory access instruction. These metrics are presented to the user along with the memory access metrics generated by the memory simulator (in the next step).

After a specified number of events has been logged or a time threshold has been reached, instrumentation is removed and the target continues executing without overhead. The compressed partial event trace is then used offline for incremental cache simulation. The cache simulator driver reverse maps addresses to variables in the source, using information extracted by the controller program, and it tags accesses to source code locations (source_filename::line_number). In addition to summary level information, the cache simulator generates detailed evictor information for source-related data structures. This information is presented to the user, along with the per-reference regularity metrics calculated by the compression algorithm.

For relating memory statistics to source code, we exploit source-related debugging information embedded in binaries. The application must provide the symbolic information in the binary (*e.g.*, generally by using the `-g` flag when compiling). Most modern compilers allow inclusion of symbolic information even if compiling with full optimizations (gcc and xlc on our platform).

2.5 Trace Generation and Compression

A large number of memory accesses can be generated within a short duration of monitoring, especially for memory-intensive codes. This access trace needs to be efficiently compressed before committing to stable storage. In addition, our compression algorithm maintains metrics describing the regularity of the access stream seen at each particular

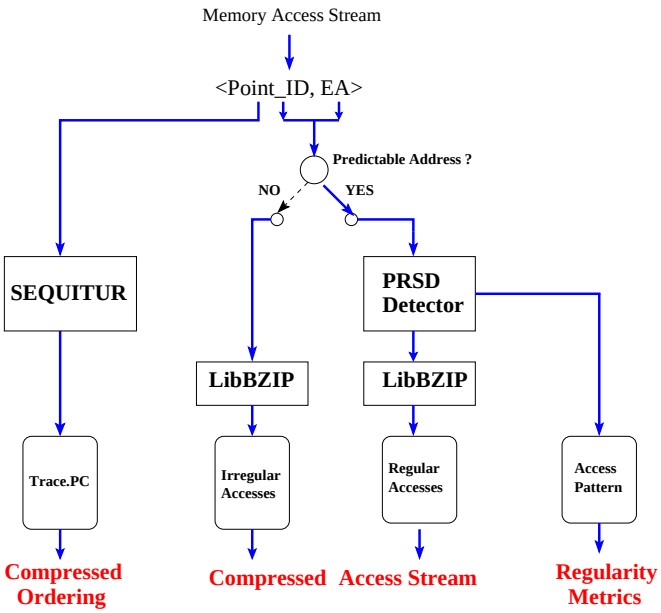


Figure 2.2: Overall Compression Algorithm

access point. These metrics provide key information during the analysis phase.

With this work we target scientific applications that tend to have highly regular accesses, usually in nested loops. We tailor our compression algorithm for this scenario. Our compression strategy is shown in Figure 2.2. The access stream to be compressed consists of individual records described by the tuple <*point_id, EA*>. `Point_id` denotes the access instruction and `EA` is the data address generated by the instruction. The task of compression is split into two parts. The *ordering* among the different access instructions is compressed separately from the *data address* generated by the individual access instructions. The idea is to use different compression algorithms suited to these distinct tasks to achieve more effective compression. It is necessary to record the access ordering for correct memory hierarchy simulation during the later phases.

2.5.1 Compressing Access Ordering

For applications with nested loops, the memory access instructions in the loop are executed in a very regular and predictable order. To exploit this regularity, we use the SEQUITUR compression algorithm to compress the IP/PC of such memory references.

SEQUITUR is described by Nevill-Manning and Witten [78]. It converts a trace of symbols into a context-free grammar, and has time overhead linear in the number of symbols in the trace [79]. The expansion of the grammar can be used to regenerate the original trace. SEQUITUR requires memory proportional to the total number of symbols occurring in the grammar. Since the total number of unique instruction addresses in the trace is usually small compared to the total program size, SEQUITUR is well suited for our purpose. We have observed extremely high compression rates with SEQUITUR on the SPEC2K FP benchmarks. In addition, decompression can proceed *incrementally*, *i.e.*, compressed traces can be used directly for cache simulation without an intermediate trace expansion step.

2.5.2 Compressing Trace Accesses

The accesses generated by each access point, *i.e.*, the data addresses of memory references, are compressed separately. In other words, our compression scheme exploits the *local* value locality of each access point. The compression algorithm is tailored for regular accesses generated by tightly nested loops. The basic unit of representation for the compressed stream is the *regular section descriptor* (RSD), an extension of Havlak and Kennedy's RSDs [38]. Each RSD is a tuple <*point_id, start_address, length, address_stride*>. Intuitively, each RSD compactly represents a stream of regular accesses generated at a given access point. The `point_id` is the access point generating this RSD. The `start_address` denotes the starting address of the stream, the `length` indicates the number of accesses in the RSD. The `address_stride` denotes the change in addresses between successive addresses in the RSD. The stride of RSDs may be an arbitrary function. We restrict ourselves to constants in this work since we require fast online techniques to recognize RSDs. In different contexts, one may want to consider linear functions or higher order polynomials. Recurring references to a scalar or to the same array element map to RSDs with a constant address stride of zero. An example RSD is shown in Figure 2.3, assuming each array element has size one.

RSDs are only sufficient to describe accesses generated by a single innermost loop. In order to efficiently describe accesses by a *nest* of loops, we introduce the *power* regular section descriptor (PRSD). A PRSD is described by the tuple <*point_id, start_address, length, address_stride, child_RSD*>. A PRSD is similar to an RSD, but instead of generating *addresses*, it generates instances of PRSDs or RSDs. The address_stride of the PRSD represents the difference in addresses between the starting addresses of two consecutive child

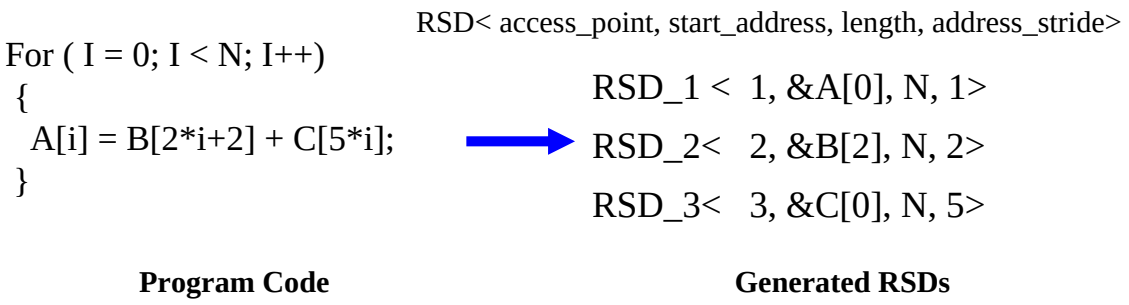


Figure 2.3: Example RSDs

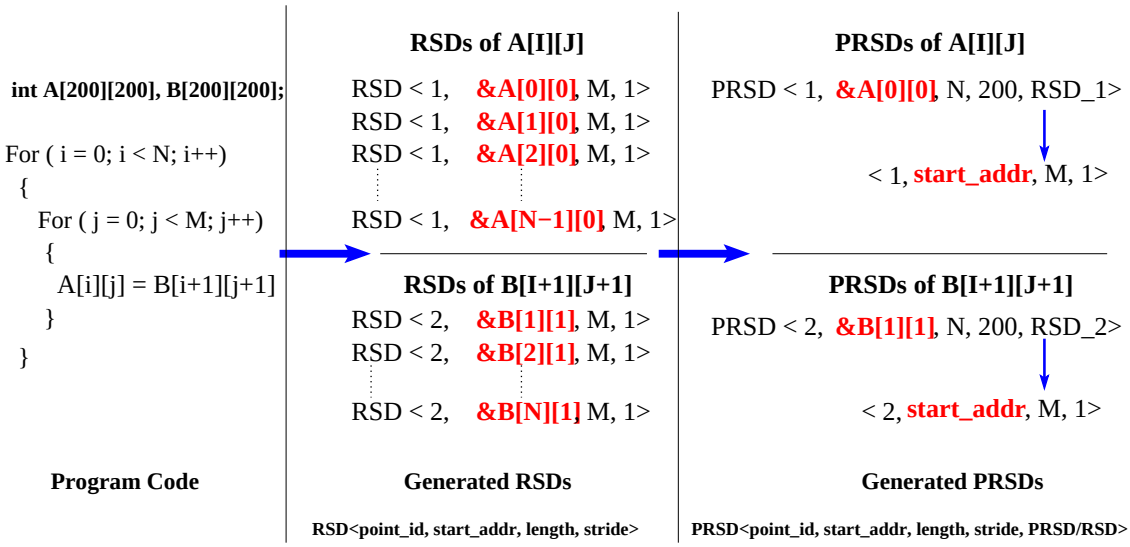


Figure 2.4: Example PRSDs

PRSD/RSDs. Thus the recursive structures of the PRSD allows efficient representation of regular accesses generated in tight loop nests.

An example PRSD is shown in Figure 2.4, assuming the size of integers is one and arrays are laid out in row major order. The RSDs for the A[i][j] and B[i+1][j+1] access points are calculated separately. There are N RSDs for each access point, each corresponding to one iteration of the outer i loop. These RSDs are compactly represented by the PRSDs shown on the right side. For example, consider the PRSD for access point of A[i][j]. The PRSD has length N, the length of the outer loop. The address stride of the PRSD is 200, since the starting addresses of A[i][j] in consecutive iterations of the i loop differ by 200.

Each instance of the PRSD is an RSD that has M elements and an address stride of one. This RSD describes all iterations in the inner j loop. The compression of data accesses proceeds as follows. The PRSD detector checks whether the incoming data access is predictable by a PRSD/RSD. If the access is predictable, the PRSD/RSD data structures are updated. Accesses may cause evictions of currently existing PRSDs/RSDs (as described in the next section). These evicted PRSDs/RSDs are further compressed by a second stage compressor based on the open source BZIP2 package [94]. BZIP2 compresses using a block sorting algorithm described by Burrows and Wheeler [15].

RSDs with less than three elements are considered irregular accesses. Irregular accesses are compressed by a separate instance of the BZIP2-based second stage compressor. In addition to compression, the PRSD detector also computes metrics characterizing the regularity of the data accesses generated by each access point. These metrics are presented in later sections and help in deeper understanding of the program's memory access behavior.

2.6 Online Detection of PRSDs and RSDs

In this section we introduce our algorithm for efficient detection of PRSDs and RSDs from the data access stream generated at each access point. To simplify the notation, we consider RSDs to be a special instance of PRSDs in the description of the algorithm. The *height* of the PRSD denotes the number of child RSDs encapsulated by the PRSD, and indicates the degree of hierarchy of the PRSD. RSDs have height zero (since they themselves do not have child RSDs).

The overall algorithm has been previously described in my Master's thesis [62]. However, we have modified that algorithm to only compress the trace *addresses* and not the trace *ordering*. The ordering is now compressed separately by SEQUITUR. As a result, we have removed the concept of explicit sequencing (sequence values and strides) from the original algorithm. In addition, the description has been revamped for readability.

The algorithm is intuitive. The algorithm builds up hierarchical structures (*i.e.,* PRSDs) as data accesses are generated at the access point. If a PRSD exists for the access point, and it can predict the incoming data access, then the PRSD length is simply incremented, and processing ends. Changes in the access stream (*e.g.*, the beginning of a new loop iteration) can cause the current PRSD to fail to predict the incoming access. This

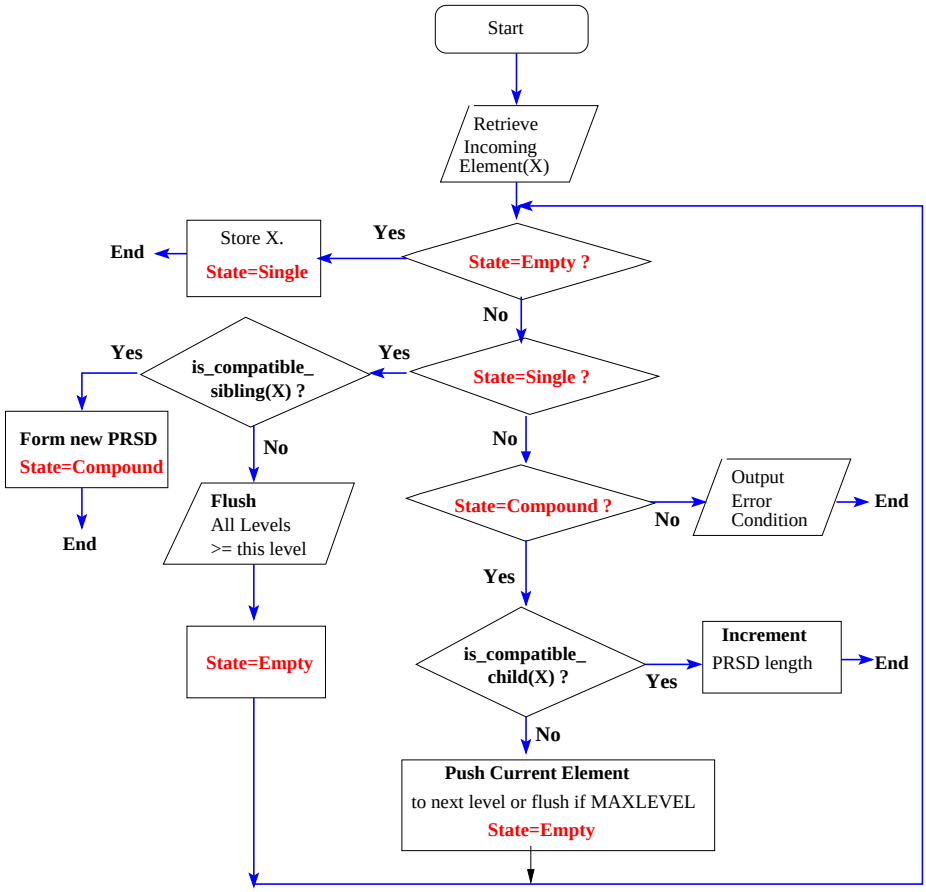


Figure 2.5: PRSD Detector Flowchart: Processing in a Level [62]

triggers formation of a new PRSD, and potentially *flushes* the current PRSD to the output buffer.

2.6.1 Levels

For each access point, we maintain a list of numbered *levels*. Each level contains a single PRSD. Higher-numbered levels contain more deeply nested PRSDs, *i.e.*, PRSDs with increasing heights. The current data access to be compressed is processed at the lowest level, *i.e.*, level zero. This may trigger the movement of any existing RSD at level zero to the next level, which may trigger the upward movement of PRSDs to higher-numbered levels.

Each level is always in one of three states — *empty*, *single* or *compound*. A level in state *empty* has no PRSDs. Similarly, a level in state *single* has only a single PRSD. A level in state *compound* has a composite PRSD. The idea is that an incoming PRSD at this level would be checked against the composite PRSD to see if it qualifies as a "child" of the composite PRSD. If so, we only need to increment the length of the composite PRSD by one — the incoming PRSD was expected. For streams with long regular accesses, we expect the level to be in the *compound* state for long stretches of processing.

2.6.2 Per-level Processing

Figure 2.5 shows the processing at each level. All levels are initially empty. Let `X` denote the incoming element to be processed at the current level number. As described earlier, the data access to be compressed is processed at level zero. Thus, `X` for level zero will be simply a data address. At higher-numbered levels, `X` will be a PRSD.

The processing of X is determined by the current state of the level. If the level is *empty*, the incoming element is simply stored, the level state is changed to *single* and processing ends.

If the level is in state *single*, there already exists a PRSD "`Y`" at this level. We try to combine the incoming element `X` with the current element `Y` to form a more deeply nested PRSD with a height equal to the height of `Y` plus one. This checking is done by the function `is_compatible_sibling`. Two PRSDs are compatible if they have the same height, length and their children are also compatible with each other (checked recursively by `is_compatible_sibling`). If the elements are compatible, a new PRSD ("composite PRSD") is formed with length two and a height equal to the height of `Y` plus one. This new PRSD will have the start_address same as the start_address of `Y` and an address_stride of the difference between the start_addresses of `Y` and `X`, and it will encapsulate `Y` as the child_prsd.

If `X` and `Y` are not compatible siblings, a change in the data access pattern is detected, *e.g.*, caused by a phase change in the program. We then flush all PRSDs in the current and higher-numbered levels, reset the level state to *empty* and resume processing. In this manner, phase changes are gracefully detected and handled.

Finally, the level might be in the *compound* state, indicating the presence of a composite PRSD "`Y`". If so, we check if the incoming element `X` can be considered a child of

this PRSD. This check is performed by the `is_compatible_child` function. The function first checks if `X` is a compatible sibling of the *children* of `Y`, using the `is_compatible_sibling` function introduced before. Next, the function checks if the start_address of `X` is equal to `Y`.start_address + `Y`.length * `Y`.address_stride, *i.e.*, if `X` is the next instance of the PRSDs produced by `Y`. If `is_compatible_child` succeeds, we simply increment the length of `Y` and processing ends.

If `X` is not a compatible child of `Y`, we push `Y` to the next level (where it is processed according to the flowchart), reset the level state to *empty*, and restart processing at this level with `X` again. The idea is that with future accesses, `X` might form a new PRSD `Z` that is compatible with `Y`. `Z` will be compared to `Y` when `Z` is pushed to the next level (If this new PRSD `Z` is still incompatible with `Y`, the flowchart illustrates that this will cause `Y` to be flushed). With access points in a recursive function, the number of levels is potentially unbounded. To guard against this, we specify a MAXLEVEL constant value beyond which the element being pushed is simply flushed to the output buffer, rather than being re-processed at a higher level.

2.6.3 Example

Figure 2.6 shows the operation of the PRSD detection algorithm for the `A[i][j]` reference shown in Figure 2.4. The figure shows the accesses generated at different instances of the loop nest, the expected actions that the algorithm executes, and the state of the data structures after these actions.

Let us step through some of the frames in the example. For each frame, we show the value of the loop index variables `i` and `j` and the corresponding memory address generated, which is input to the PRSD detection algorithm.

Frame 1: This shows the initial state. All the levels are in state *empty*.

Frame 2: (`i=0, j=0`, &A[0][0]): This is the first iteration point in the loop nest. The incoming element is stored in level zero and the state of the level is changed to *single*.

Frame 3: (`i=0, j=1`, &A[0][1]): The incoming element and the resident element are compared to verify that they can be combined into a composite PRSD (`is_compatible_sibling`). The new composite PRSD has length two, and the state of the level is updated to *compound*.

Frame 4: (`i=0, j=2`, &A[0][2]): The incoming element is checked to verify that

<table>
<tr>
<td>Input: (Iter, Jiter), address
Action: What steps to take for this input element.

int A[200][200]
For (i=0; i < N;i++)
{
For(j=0; j < M;j++)
{
A[i][j] = ...
}
}</td>
<td>(1) Input: None (initial State)
Action: None. All levels are empty.

Level 0. State= Empty

Level 1. State= Empty</td>
<td>(2) Input: (i=0,j=0) &A[0][0]
Action: Store element in level−0.
Update level−0 state

Level 0. State= Single
< &A[0][0] >

Level 1. State= Empty</td>
</tr>
<tr>
<td>(3) Input: (i=0,j=1) &A[0][1]
Action: is_compatible_sibling? YES
Form composite PRSD.

Level 0. State= Compound
RSD: < Start_addr= &A[0][0],
Addr_stride = 1
Length = 2 >

Level 1. State= Empty</td>
<td>(4) Input: (i=0,j=2) &A[0][2]
Action: is_compatible_child ? YES
Increment PRSD length.

Level 0. State= Compound
RSD: < Start_addr= &A[0][0],
Addr_stride = 1
Length = 3 >

Level 1. State= Empty</td>
<td>(5) Input: (i=0,j=N−1) &A[0][M−1]
Action: is_compatible_child ? YES
Increment PRSD length.

Level 0. State= Compound
RSD: < Start_addr= &A[0][0],
Addr_stride = 1
Length = M >

Level 1. State= Empty</td>
</tr>
<tr>
<td>(6) Input: (i=1,j=0) &A[1][0]
Action: is_compatible_child ? NO !
Push PRSD to level−1.
Re−process incoming element

Level 0. State= Single
< &A[1][0] >

Level 1. State= Single
RSD: < &A[0][0], 1, M−1></td>
<td>(7) Input: (i=1,j=1) &A[1][1]
Action:
Level−0: is_compatible_sibling? YES
Form composite PRSD.

Level 0. State= Compound
RSD: < Start_addr= &A[1][0],
Addr_stride = 1
Length = 2 >

Level 1. State= Single
RSD: < &A[0][0], 1, M−1></td>
<td>(8) Input: (i=1,j=M−1) &A[1][M−1]
Action:
is_compatible_child ? YES
Increment PRSD length.

Level 0. State= Compound
RSD: < &A[1][0], 1, M>

Level 1. State= Single
RSD: < &A[0][0], 1, M></td>
</tr>
<tr>
<td>(9) Input: (i=2,j=0) &A[2][0]
Action:
1. Push RSD to level−1
2. Level−1: is_compatible_sibling? YES
3. Level−1: Form composite PRSD
4. Level−0: re−process incoming element

Level 0. State= Single
< &A[2][0] >

Level 1. State= Compound
PRSD:<
Start_addr= &A[0][0]
Addr_stride = 200, Length = 2,
Child_RSD=<−,stride=1,length=M></td>
<td>(10) Input: (i=3,j=0) &A[3][0]
Action:
1. Push RSD to level−1
2. Level−1: is_compatible_child? YES
3. Level−1: increment PRSD length
4. Level−0: re−process incoming element

Level 0. State= Single
< &A[3][0] >

Level 1. State= Compound
PRSD:<
Start_addr= &A[0][0]
Addr_stride = 200,
Child_RSD=<−,stride=1,length=M></td>
<td>(11) Input: (i=N−1,j=M−1) &A[N−1][M−1]
Action:
1. Level−0: is_compatible_child ? YES

This is how data structures look after last access in loop nest.

Level 0. State= Compound
RSD: < &A[N−1][0], 1, M>

Level 1. State= Compound
PRSD:<
Start_addr= &A[0][0]
Addr_stride = 200, Length = N−1,
Child_RSD=<−,stride=1,length=M></td>
</tr>
</table>

Figure 2.6: PRSD Detection Example

it can be considered to be a *child* of the currently resident composite PRSD (`is_compatible_child`). The length of the composite PRSD is incremented by one and processing ends.

Now we skip to the last iteration of the i loop in the same iteration of the i loop.

Frame 5: (`i=0, j=M-1`, &A[0][M-1]): The incoming element qualifies as a child of the resident PRSD (`is_compatible_child`).The length of the resident PRSD is incremented by one and processing ends.

Frame 6: (`i=1, j=0`, &A[1][0]): This is the very next iteration point of the loop nest after Frame 5 and is the first access in iteration 1 of the i loop. Assuming `M` is smaller than 200 (the lower dimension of the array), the currently resident PRSD will not correctly predict the incoming element (the PRSD will predict address &A[0][M+1], the incoming address is &A[1][0]), *i.e.*, `is_compatible_child` will fail. The currently resident PRSD is pushed to the next level, and the incoming element is saved in the current level.

Frame 9: (`i=2, j=0`, &A[2][0]): This is the next iteration point after Frame 8. The incoming element will not be predicted by the currently resident PRSD on level zero (similar to Frame 6), which will cause the PRSD to be pushed to the next level (level one). In level one, this PRSD is compared to the pre-resident PRSD to verify that they are compatible siblings (`is_compatible_sibling`), after which a new composite PRSD is formed with length two, as shown. The state of level zero is reset to *empty* and processing is restarted with the incoming address &A[2][0].

Frame 10: (`i=3, j=0`, &A[3][0]): Similar to Frame 9, the resident PRSD at level zero will not be able to predict the incoming address &A[3][0]. This will cause the resident PRSD to be pushed upwards to level one, where it will qualify as a child of the pre-resident PRSD. This will cause the length of the pre-resident PRSD at level one to be incremented by one, as shown.

Frame 11: (`i=N-1, j=M-1`, &A[N-1][M-1]): This is the last access of the loop nests. The incoming element will be correctly predicted by the currently resident PRSD in level zero (similar to Frame 5). The state of the data structures at the end of this access is as shown — there is an RSD at level zero and a PRSD at level one. Future accesses at the current access point will cause the RSD to be pushed to level one where it will qualify as a child of the pre-resident PRSD.

2.6.4 Space Complexity

In the worst case, a completely random sequence of addresses can be passed to the PRSD detection algorithm. In this case, no RSDs or PRSDs will be detected and the accesses will be recorded individually as irregular accesses. Thus, the space complexity of the algorithm is O(M), where M is the total number of accesses (*i.e.*, linear space complexity). The best case input is a stream of regular accesses. For such input the algorithm would, at best, generate exactly one PRSD for each access point. The space required to represent a PRSD is proportional to its *height*. The height of the PRSD in a particular level can be at most one greater than the level number, which has an upper bound given by the constant value MAXLEVELS. Thus, the space complexity to represent the PRSDs for `n` access points is bounded as O((MAXLEVELS+1)*n). `n` is an attribute of the source code and is constant for the duration of monitoring. Since both factors are constant, the best case space complexity has a constant upper bound.

2.6.5 Time Complexity

Since we must look at each incoming element to compress it, the lower bound on the time complexity is given as Ω(M), where M is the total number of accesses in the trace. A particular incoming access may trigger movement of PRSDs/RSDs to higher-numbered levels, where they need to be re-processed. The number of re-processing steps is bounded by the maximum number of levels (MAXLEVELS) and the height of the PRSD, which can be at most (MAXLEVELS+1). Thus, the upper bound on time complexity is O(M*MAXLEVELS*MAXLEVELS). Since MAXLEVELS is constant, the upper bound on the time complexity is linear in the number of accesses in the trace.

2.7 Evaluation of the Compression Scheme

In this section, we evaluate the performance of our compression scheme with respect to compression efficiency and time required for compression. We compare our results for 12 out of the 14 SPEC2000FP benchmarks [1]. Results are compared against VPC3, a

[1] `191.fma3d` failed to run because DynInst ran out of memory for instrumentation code. `301.apsi` failed due to an internal error in DynInst.

state-of-the-art compression algorithm based on using value predictors for data compression [16].

2.7.1 VPC3

VPC3 is targeted for compression of extended address traces. Such traces contain the instruction address (PC) of the access instruction, followed by one or more register values or effective addresses (EA). VPC3 first splits the access stream into separate streams of PCs and EAs. The algorithm has a bank of value predictors that attempt to predict the target element value (PC or EA). All predictors are updated after each element has been processed. VPC3 by itself does not compress the trace. Instead, it writes out the *id* of the value predictor that successfully predicted the current element. This stream of *ids* is compressed by a second stage compressor based on BZIP2. Elements that were not predicted by any predictor are compressed by a separate instance of the second stage compressor. In our experiments, we use the VPC3 source code obtained from the author's website [17] and couple the output to a second stage compressor based on BZIP2 [94].

We use VPC3 for comparison since it represents the state-of-the-art in compressing access traces. VPC3 has been shown to compress faster and with more effective compression rate for most benchmarks, compared to several contemporary compression algorithms (SEQUITUR, BZIP2, GZIP) [16]. VPC3 is targeted towards efficiently compressing the address traces of general purpose programs while we focus specifically on programs found in scientific computing. However, in addition to compressing access traces, our approach generates metrics that *characterize* the address stream (described later in Section 2.11). These metrics, along with the results generated by the simulator, provide insight into the application's memory access behavior.

2.7.2 Experimental Setup

For our compression scheme we used the open source implementation of SEQUITUR [61]. All benchmarks were compiled at -O2 optimization level on an IBM POWER4 platform. All benchmarks used "training" data sets. The static call graph of the target program was traversed with `main` as root, and all memory access points in the call graph were instrumented. Up to one billion (10^9) accesses were traced and compressed on-

Table 2.1: Comparison of Compression Rates

Benchmark	Our Algorithm	VPC3	Ratio: (Ours) / VPC3
171.swim	**910608.59**	154698.98	**5.886**
168.wupwise	144.74	221.48	0.653
172.mgrid	**70847.45**	4765.63	**14.866**
173.applu	**337.52**	133.94	**2.519**
177.mesa	1519.42	6183.17	0.245
178.galgel	1938.03	4466.73	0.433
179.art	**283312.87**	40380.65	**7.016**
183.equake	12.23	99.55	0.122
187.facerec	**2382.55**	618.93	**3.849**
188.ammp	**1496.68**	1152.85	**1.298**
189.lucas	**607.34**	437.52	**1.388**
200.sixtrack	181.24	488.11	0.371
Geometric_Mean	**2196.76**	1636.89	
Harmonic_Mean	118.76	407.20	
Average	**106115.72**	17803.97	

line for each benchmark. All benchmarks reached the one billion limit, except for 177.mesa ($8x10^6$ total accesses) and 188.ammp ($531x10^6$ total accesses).

2.7.3 Comparison of Compression Rates

The compression rate was computed as follows. The uncompressed access trace is composed of <point_id, address> records. Each uncompressed record requires six bytes — four bytes for the 32-bit address and two bytes for the point_id. Notice that all our programs had less than 65536 memory access points. Thus the total uncompressed trace size is (# total_records) * 6. The compression rate is calculated as $\frac{size\ of\ un-compressed\ trace}{size\ of\ compressed\ trace}$.

Table 2.1 shows the compression rates for our algorithm and for VPC3. The last column shows the *relative* compression rate of our algorithm compared to VPC3. The table shows that both VPC3 and our algorithm achieve substantial compression rates on almost all the benchmarks. For 7 out of the 12 benchmarks, our algorithm achieves a better compression rate than VPC3 (boldface ratio in last column greater than one). For some programs with very regular loop nest oriented structures, our algorithm achieves spectacularly large compression rates (`swim, mgrid, art`), due to our use of hierarchical PRSD structures. Overall, the geometric mean of the compression rate of our algorithm is

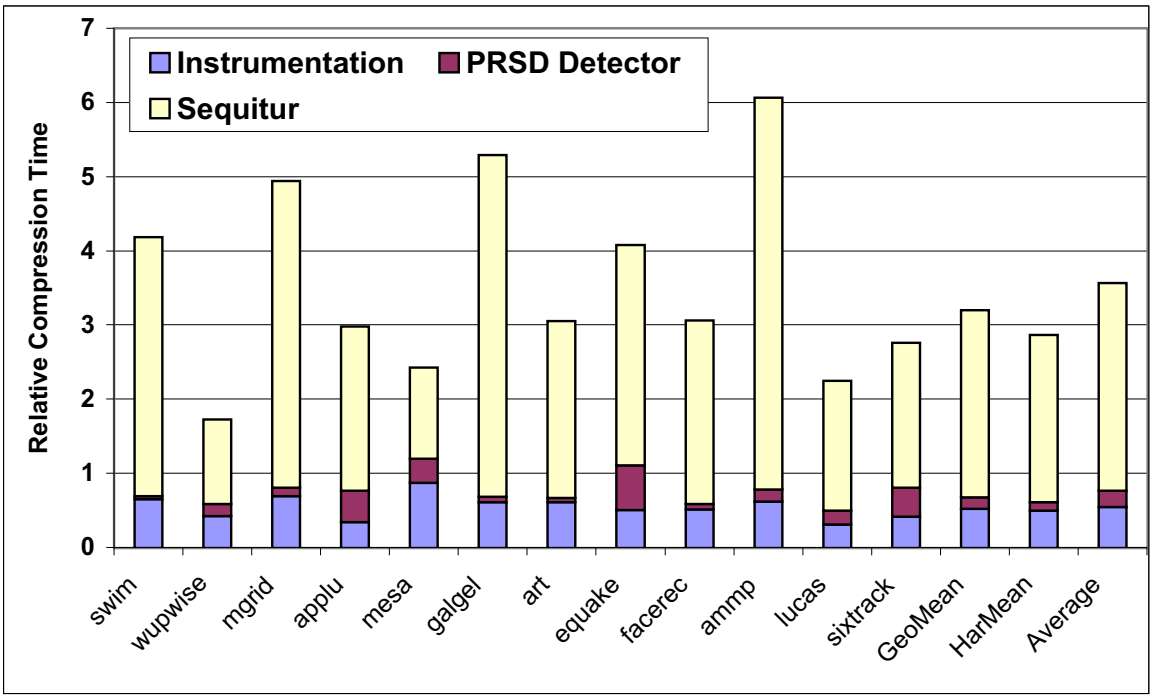


Figure 2.7: Execution Time Breakup for Our Compression Scheme, Relative to VPC3 Execution Time

about 25% greater than the value for VPC3.

2.7.4 Comparison of Compression Times

Figure 2.7 shows the time required for compression using our algorithm. The time for three different components is shown. *Instrumentation* denotes the overhead of the binary instrumentation framework (*e.g.*, saving/restoring register context). *PRSD Detector* denotes the overhead of the PRSD detection algorithm introduced in the last section. *Sequitur* denotes the overhead of the SEQUITUR-based compression of the trace ordering. The values are relative to the time taken by the VPC3-based online compression framework (including instrumentation overhead, which should be similar in both cases). Our algorithm is on average three times slower than the VPC3 implementation. By far the most expensive component is the SEQUITUR-based module for compressing the trace ordering. It may be possible to reduce this overhead by using a more optimized version of SEQUITUR. Alternately, we could update the stride predictor in VPC3 to use PRSDs. This modified VPC3 would be much faster than our current approach while allowing us to leverage VPC3's com-

pression capabilities on programs where the accesses are less regular. However, we would lose structural information inherent to PRSDs after BZIP compression. Nevertheless, the PRSD predictor would still generate the *regularity* metrics (discussed later in Section 2.11) that complement the results generated by the memory hierarchy simulator. Finally, we note that METRIC is capable of and intended for gathering *partial* access traces, where the overhead of trace compression is limited by the duration of monitoring. Thus, in practice, a slightly more expensive scheme might still be acceptable as long as the trace collection period is short.

2.8 Memory Hierarchy Simulation

The compressed trace obtained in the preceding sections is used offline for incremental memory hierarchy simulation. After a partial trace of accesses has been collected, the instrumentation is removed dynamically and the application continues execution without overhead. For programs that exhibit distinct phases of execution (*e.g.*, time-stepped programs), this allows us to limit the overhead of performance analysis by capturing and simulating only "snippets" of the complete trace.

For memory hierarchy simulation, we use a modified version of `MHSim` [70]. MHSim simulates the data TLB (translation Lookaside Buffer) and multiple levels of cache. MHSim maintains information *per-reference*, allowing "bulk metrics" regarding memory performance (*e.g.*, hits, misses) to be drilled down and mapped to individual access points. For each access point, it generates a rich set of metrics that we shall discuss further below. The original MHSim package used a source-to-source Fortran translator to annotate data accesses with calls to MHSim cache simulation routines. This strategy has two significant disadvantages, which we overcome with our approach.

The most serious problem with source instrumentation is that it may significantly distort the actual memory access behavior of the program without instrumentation. Annotating the source code accesses with function calls to MHSim routines will potentially inhibit many important and well established loop reordering transformations (*e.g.*, loop interchange, tiling), because of the additional true dependences introduced by the function calls. It may also prevent or modify other standard compiler optimizations, such as common sub-expression elimination (due to presence of function calls accepting addresses of array

references). Thus, the resultant executable with instrumentation can be totally different (in terms of memory access patterns) from the original uninstrumented version — which can lead to potentially misleading diagnostic information reported by MHSim. In contrast, by instrumenting the final optimized binary generated by the compiler, we guarantee that we still capture the exact original access pattern. Thus, we can generate diagnostic information that correctly reflects the target program behavior. Consequently, we argue that source-level instrumentation is the *wrong abstraction level* for capturing the original application behavior and can lead to potentially misleading results for programs in our target domain (loop-oriented scientific codes).

The second major problem with source-level instrumentation frameworks is that they are limited to a particular language. Many scientific programs are mixed-language applications [108]. In addition, many programs make heavy use of libraries (*e.g.*, Standard C library (libc), math and numerical libraries, networking libraries), that a source level instrumentation frameworks will be unable to instrument. Thus the resultant trace of memory accesses may be incomplete and can lead to potentially misleading diagnostic information. In contrast, our approach is independent of any language, compiler and linker. More importantly, we use *dynamic* binary rewriting that allows us to instrument target applications as they are executing. Thus, we can turn the instrumentation on and off, enabling the capture of *partial* access traces as discussed before. The resulting overhead of trace collection and instrumentation is flexible and is only limited to the duration of monitoring.

2.9 Abstracting Trace Data

The compressed trace contains "raw" instruction addresses (point_ids) and data addresses. We use the symbolic information embedded in the binary to map the instruction addresses to source code locations (filename::line_number). We also try to reverse map the raw data address to a symbolic variable name using information extracted from the embedded symbol table. Global variable names and sizes are easily obtained from the symbol table. We also support local variables by keeping records of function entry and exit in the trace, and by recording the value of the stack pointer on entry. The symbol table for local variables only contains the address *offsets* in the current activation record of the function. Combined with the value of the stack pointer recorded in the trace, this

allows us to reverse map accesses to function-local variables. Finally, dynamically allocated variables can be partially supported by instrumenting the entry to allocation functions (malloc/calloc/free) and walking the call stack at allocation to create a unique "allocation context" identifier. The data accesses to elements in the dynamically allocated area will be reverse mapped and tagged to this identifier in the MHSim report.

2.10 MHSim-generated Metrics

MHSim generates metrics for each level of cache and also for the data TLB. Metrics can be aggregated by reference, by variable and by loop nest. We shall list and describe each metric and later discuss their value as diagnostic input to understand memory behavior. MHSim generates the following metrics per-reference:

- **Hits**: Number of accesses by this reference point that hit in the cache.
- **Misses**: Number of accesses by this reference point that missed in the cache.
- **Miss Ratio** Ratio of hits to misses.
- **Temporal Hit Fraction**: The fraction of the hits that occurred due to *temporal reuse* of data. Calculated as $\frac{temporal\ hits}{total\ hits}$. MHSim uses bit vectors to maintain information about which byte offsets in the cache line were addressed by access instructions, allowing classification of hits into temporal and non-temporal hits. Temporal hits include hits caused by both *self-reuse* (same reference point accesses a memory location multiple times) and *cross-reuse* (different reference points access same memory location).
- **Spatial Hit Fraction**: This is defined as 1 - temporal_hit_fraction, *i.e.*, non-temporal hits are classified as purely spatial hits.
- **Spatial Reuse**: This value gives the average fraction of the memory line in bytes that was *used*, *i.e.*, explicitly addressed by a memory access instruction, before the memory line was evicted from the cache. It is computed as $\frac{used\ bytes}{cache\ line\ size * number\ of\ evictions}$.
- **Evictor References**: For each reference, MHSim maintains a list of *evictor* references that evicted this reference from the cache. Evictors provide insight into cache conflicts. Cycles of evictors potentially indicate conflict misses which could be removed by transformations like padding.

2.11 Stream-oriented Metrics

In addition to the metrics generated by MHSim, the PRSD detector in the compression algorithm also generates complementary metrics characterizing the regularity of the access stream. These metrics are calculated separately for each access point. The following metrics are generated:

- **Regularity ratio**: Computed as $\frac{total\ predictable\ accesses}{total\ accesses\ at\ this\ point}$. Predictable accesses are those detected as an instance of an RSD or PRSD. The regularity ratio allows us to classify access points into irregular and regular categories. Access points with high regularity ratios can be targeted for stream-based optimizations, as described in our previous work [71]. For example, the predictable nature of the access point can be exploited by prefetching, which caches future data access early to lessen effective access latencies.
- **Mean stream length**: The average of the length of all RSDs generated at this point.
- **# Distinct lengths**: Number of distinct RSD lengths seen at this access point.
- **% Distribution of distinct lengths**: The distribution of RSDs according to their lengths.
- **# Distinct strides**: Number of address strides for all RSDs seen at this point.
- **% Distribution of distinct strides**: The distribution of RSDs according to their address strides.

The definition of regularity ratio as defined here differs from the definition in our previous work [71]. In our previous work, the regularity ratio was a single value calculated over the *entire* program or program section to characterize the stream behavior. Access streams were not segregated by access point, *i.e.*, a stream could contain accesses from different access points. In contrast, in this work we segregate the access stream by access points and calculate the regularity metrics for each point separately. Thus, we can now obtain a much finer level of information tagged to individual access points, instead of a single aggregate value for the whole program or program section.

Metric	Diagnostic Information
Miss Ratio	A basic measure of performance. References with high or medium miss ratios should be specifically singled out for further analysis. A high miss ratio, when other indicators like regularity ratio and stream lengths have favorable values, indicates presence of specific cache access inefficiencies.
Temporal Hit Fraction	This measures how much temporal reuse is being realized for the memory lines accessed by this reference. Low value may indicate that the reference is being flushed from cache before reuse could occur. If low temporal reuse is inherent to the reference, *cache hinting* can be used to avoid allocating a cache line (this requires other indicators to show specific behavior, see text for use case).
Spatial Reuse	Low values indicate that cache is not being used efficiently — data is being brought in which is never "touched" before the memory line is evicted from cache. Can indicate presence of conflict misses, if regularity metrics (regularity ratio, stride values) show regular and low-strided access behavior.
Evictor References	A cycle of evictors coupled with other indicators like low spatial reuse can indicate presence of conflict misses. The advantage of evictor references is that it tells us *precisely* which references are involved in the conflict, allowing straightforward code/data transformations to correct it. On the other hand, when other indicators of cache efficiency (*e.g.*, spatial reuse) are high, cycles of evictors may still indicate the presence of *capacity* misses — there is simply not enough room in the cache to keep all the accessed data at the same time.
Regularity ratio	Highly regular streams produce *predictable* values, which can be exploited by optimizations like prefetching. On the other hand, irregular references can be optimized by another class of optimizations (*e.g.*, cache hinting). References with high regularity ratios that still have high miss rates reveal the presence of cache access inefficiencies.
Mean stream length	Optimizations like prefetching require a minimum stream length to be profitable.
% Distribution of strides	Low-strided references should be expected to have high spatial reuse values, otherwise a cache access inefficiency is indicated. If there are only a few dominant strides, it may simplify the implementation of optimizations like prefetching (knowing dominant stride value allows manual insertion of prefetch instructions without depending on the compiler.)

Figure 2.8: Use of Metrics for Performance Diagnosis

2.12 Diagnosis of Performance Problems

In previous paragraphs, we introduced several metrics to quantify different facets of memory access performance. What diagnostic information do these metrics provide? How can we use them to understand the symptoms and the underlying causes of memory access inefficiencies? Figure 2.8 gives a short overview of how the generated metrics can be used for this task.

METRIC gives insight on the memory access patterns of the target program. The information provided by METRIC allows the program analyst to focus on the bottleneck of the program, and also gives indications on how a bottleneck can be removed by manually applying program or data transformations. Many of these transformations can also be achieved by contemporary compiler technology. Such transformations were presented in our earlier work for some well known computation kernels [65]. This work will not reiterate them. Instead, we shall use METRIC to optimize several sample codes to illustrate its potential advantage over compile-time analysis, particularly when interprocedural analysis is required. For clarity of presentation, the sample codes are microbenchmarks that manifest a particular performance weakness. They represent behavior that can arise in larger real world programs.

2.12.1 Use case: Cache Reuse Hinting

Consider the following snippet of C code:

```
1  double A[MATDIM], B[MATDIM];
2  double C[MAT2], D[MAT2];
3
4  void do_sum()
5   {
6      for(i=0;i < MATDIM;i++)
7              A[i] = A[i] + B[i];
8   }
9
10 void do_mult(void)
11  {
12     for(j=0;j < 1500;j++)
13             C[ind[j]] *= D[ind[j]];
14  }
15
```

```
16  void main()
17  {
18     for(i=0;i < timesteps;i++)
19      {
20            do_sum();
21            do_mult();
22      }
23  }
```

There are four distinct arrays `A`, `B`, `C` and `D` in the first use case. The functions do_sum() and do_mult() are called once per timestep. This program was compiled and traced under our framework on a Power4 platform using the IBM xlc compiler A cache with the following parameters was simulated: cache size=256 KB, associativity=8, line size=128, writeback cache, LRU replacement policy. This configuration is similar to the L2 cache of the Itanium2 processor [46]. The per-reference results generated by the simulator are shown in Figure 2.9. Figure 2.9(a) shows the cache metrics generated by the simulator, and Figure 2.9(b) shows the stream metrics generated by the PRSD detector.

Analysis

The reference name shown in the results has the following syntax: *VariableName_Accesstype_id*. *VariableName* is the symbolic identifier that corresponds to the memory address being accessed. *Accesstype* can be either *Read* or *Write*. Finally, *id* denotes the unique numerical identifier for this access instruction in the executable code of the target. This syntax is used in all the use cases presented in this work.

The per-reference results show that different references have widely different behaviors. `D_Read_12` and `C_Read_11` have very high miss rates ($> 87\%$) while the remaining references have lower miss rates ($< 7\%$). The spatial reuse values are also much lower for `D_Read_12` and `C_Read_11`, showing that on average, only 6.3% of the memory line data that was brought into the cache by these two references was accessed before eviction. The stream metrics show that accesses by `D_Read_12` and `C_Read_11` were completely unpredictable, with a regularity ratio value of 0.0. The remaining references had completely predictable access streams (regularity ratio=1.0) and were seen to be linearly strided (a single stride of eight for reference points of type `double`, a single stride of four for reference points of type `int`). All the preceding indicators show that `D_Read_12` and `C_Read_11` generate irregular accesses

with very low cache hit rates. The evictors for each reference are shown in Figure 2.10. The figure shows that in addition to poor locality, the `D_Read_12` and `C_Read_11` references are also the top evictors for all the remaining references. Thus the references to `D` and `C` bring in data into the cache that is not reused (as indicated by their low spatial reuse values) and evict a significant amount of pre-resident data from the cache (as indicated by the per-reference evictors).

A look at the source code shows the cause of this behavior. The `D_Read_12` and `C_Read_11` references are potentially sparse indirect reads on an array, indexed by the array `ind[]`. The remainder of the read references (`A_Read_7`, `B_Read_8` and `ind_Read_10`) are all direct array accesses, with regular single strided access patterns.

File	Line	Reference	SourceRef	Hits	Misses	Miss Ratio	Temporal Ratio	Spatial Reuse
reuse.c	13	**D_Read_12**	**D[ind[i]]**	1532	13468	**0.897**	0.1566	**0.0639**
reuse.c	13	**C_Read_11**	**C[ind[i]]**	1929	13071	**0.871**	0.320	**0.0639**
reuse.c	7	**A_Read_7**	**A[i]**	96125	6275	**0.061**	0.021	**0.9867**
reuse.c	7	**B_Read_8**	**B[i]**	96135	6265	**0.061**	0.023	**0.9860**
reuse.c	13	**ind_Read_10**	**ind[i]**	14530	470	**0.031**	0.019	**0.9134**
reuse.c	13	**C_Write_13**	**C[ind[i]]**	15000	0	**0.0**	1.0	**1.0**
reuse.c	7	**A_Write_9**	**A[i]**	102400	0	**0.0**	1.0	**1.0**

(a) Per-Reference Cache Statistics

Reference	Total Accesses	**Predictable Accesses**	**Regularity Ratio**	Average Length	Distinct Strides	**% Stride Distribution**
D_Read_12	15000	**0**	**0.0**	0	0	-
C_Read_11	15000	**0**	**0.0**	0	0	-
A_Read_7	102400	**102400**	**1.0**	10240	1	**stride=8,100%**
B_Read_8	102400	**102400**	**1.0**	10240	1	**stride=8,100%**
ind_Read_10	15000	**15000**	**1.0**	1500	1	**stride=4,100%**
C_Write_13	15000	**0**	**0.0**	0	0	-
A_Write_9	102400	**102400**	**1.0**	10240	1	**stride=8,100%**

(b) Per-Reference Stream Statistics

Figure 2.9: Original Per-Reference Memory Usage Statistics

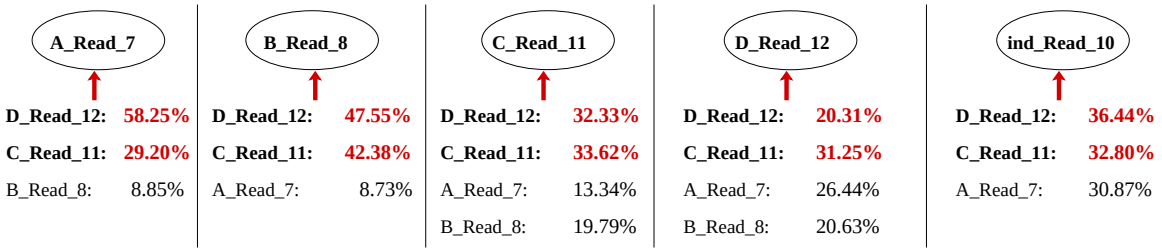


Figure 2.10: Evictors for Each Reference

Optimization

From the analysis, we know that `D_Read_12` and `C_Read_11` are the key references with a significant impact on cache performance. We also know that these references inherently have poor cache reuse, due to their irregular data access pattern. Instead of trying to reorder their access patterns, we can try to reduce their detrimental impact on the cache by asking the memory system *not to allocate a normal cache line for these references.*

This is achieved using the concept of *reuse hints.* Reuse hints are tagged to each memory reference instruction (ld/st) and provide hints to the memory subsystem on the potential reuse of the data fetched by this access instruction. The Itanium2 ISA implements such a hinting mechanism [46]. Hints indicate whether the accessed data has no expected temporal locality at the level of the L1 cache (hint=`.nt1`), at the level of the L2 cache (hint=`.nt2`) or no temporal locality at any level (hint=`.nta`). Floating-point accesses bypass the L1 cache. So, for these accesses, `.nt1` refers to the *L2* cache and there is no `.nt2` hint. For floating-point references with `.nt1` or `.nta` hints that miss in the L2 cache, the L2 cache will allocate a cache line in only one out of the eight associative ways. The data in the remaining part of the cache is undisturbed. In addition, the LRU bits in the cache are not updated, so the allocated line will soon be selected for eviction.

We test our optimization on an actual Itanium2 system. We target the L2 cache, and tag the `D_Read_12` and `C_Read_11` references with `".nt1"` hints. The hints will minimize the impact of these two references on the data pre-resident in the cache. In this way, we hope to retrieve any potential "locality" on the other references that was lost due to these two interfering references. We note that tagging `C_Read_11` but not `C_Write_13` will not

File	Line	Reference	Source Ref	Hits	Misses	Miss Ratio	Temporal Ratio	Spatial Reuse
reuse.c	7	**A_Read_7**	**A[i]**	101759	641	**0.006**	**0.905**	**1.0**
reuse.c	7	**B_Read_8**	**B[i]**	101760	640	**0.006**	**0.905**	**1.0**
reuse.c	13	**ind_Read_10**	**ind[i]**	14953	47	**0.003**	**0.902**	**1.0**
reuse.c	7	**A_Write_9**	**A[i]**	102400	0	**0.0**	**1.0**	**1.0**

Figure 2.11: Optimized Per-Reference Memory Usage Statistics

provide the desired benefit since the line would be cached following the second access. This need to tag what appears to be a well-performing access demonstrates the complexity of the analysis that would be required by a compiler. The optimized code in the do_mult() function is shown below:

```
10 void do_mult(void)
11 {
12   for(j=0;j < 1500;j++)
13   {
         index=ind[j];
         value =  read_double_nt1(&C[index])
                * read_double_nt1(&D[index]);

         write_double_nt1(&C[index],value);
       }
14 }
15
```

The `read_double_nt1` and `write_double_nt1` are special inlined functions that load and store doubles using instructions with explicit ".nt1" hints.

First, let us the see the potential impact of the optimization using the simulator. Our simulator currently does not support hinting for the access points. Instead, we run the same program again but without the `D[]` and `C[]` array accesses and see the change in cache metrics for the remaining references, as shown in Figure 2.11.

Notice the improvement in the hit rates for the `A_Read_7`, `B_Read_8` and `ind_Read_10` references as compared to the original behavior. The miss ratios for these references have decreased by an order of magnitude (*e.g.*, 6% to 0.6% for `A_Read_7` reference). The temporal fraction of the hits has gone up to 90% for these references, compared to the less than 3%

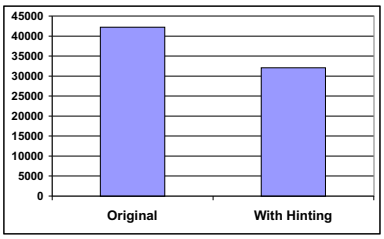


Figure 2.12: Comparison of L2 Cache Misses

in the original results. This indicates that we are now realizing *inter-timestep locality* — the data is brought into the cache during the first time step and almost always remains in cache till it is accessed again during the next time step.

Let us now test our optimization on the real system. The original program and the optimized version with cache hints were both compiled and run on an Itanium2 system. In each case, we monitor the hardware counters and count the number of L2 misses. Specifically, we measure the value of the `L2_MISSES` event for the original and optimized program. The values for the two runs are shown in Figure 2.12. The number of L2 misses reduces from 42214 in the original program to 32072 in the optimized program (a 24% reduction).

We demonstrated how METRIC can be used for setting reuse hints. It is very hard or impossible for a static compiler to perform this analysis, since the complete runtime memory access pattern of the program must be considered (*e.g.*, if the `D_Read` and `C_Read` hit in cache in the original program, the reuse hinting may actually be detrimental). The compilers evaluated (Intel icc 8.0, gcc 3.4) did not automatically set the non-temporal hints for the `D_Read` and `C_Read`. (For the optimized code, we inserted the hints manually using inline assembly functions).

2.12.2 Use case: Prefetching

Consider the following snippet of C code:

```
0 #define MATDIM 1000
1  double A[MATDIM][MATDIM], B[MATDIM][MATDIM];
2
3  void do_mult(void)
4  {
5     for(i=0;i < MATDIM;i++)
```

```
6          for(j=0;j < MATDIM;j++)
7             {
8                 A[i][j] = A[i][j] * B[j][i];
9             }
10 }
```

There are two two-dimensional arrays `A` and `B`. The function calculates the product of `A[i][j]` with `B[j][i]`, and stores the value back into `A[i][j]`. This program was compiled on a Power4

machine using xlc [2]. A cache with the following parameters was simulated: cache size=32 KB, associativity=2, line size=128, writeback cache, LRU replacement policy. This configuration is similar to the L1 cache of a Power4 processor. The simulator reported the following cache performance:

hits	=	1937499	temporal hits	=	1000000
misses	=	**1062504**	spatial hits	=	937499
temporal ratio	=	0.51613	spatial ratio	=	0.48387
miss ratio	=	**0.3541**	**spatial reuse**	=	**0.17836**

Notice the high miss ratio (**35%**) and the relatively low spatial reuse value (**17.8%**). The per-reference results are shown in Figure 2.13. Figure 2.13(a) shows the metrics generated by the cache simulator, and Figure 2.13(b) shows the stream metrics generated by the PRSD detector. Due to instruction scheduling, the compiler unrolls the very last iteration of the innermost loop, hence there are several additional access instructions present in the executable (more than the three access instructions in the original C code). For clarity of presentation, we do not show the metrics associated with these additional access points. This explains why the number of accesses for the references shown in the per-reference result do not exactly match the number of accesses expected from the C source version.

Analysis

`B_Read_3` has the worst possible cache performance — all of its accesses are misses.

[2] -O3 optimization level with loop unrolling turned off. Unrolling the loop body gives rise to many additional access instructions that show up as separate access points in the MHSim results. For clarity of presentation, we turn off unrolling the loop body so that fewer access points are present in the binary code. However, we could not prevent the compiler from unrolling the very last iteration of the inner loop, as explained in the text.

File	Line	Reference	SourceRef	Hits	Misses	Miss Ratio	Temporal Ratio	Spatial Reuse
test.c	31	**B_Read_3**	**B[j][i]**	0	999000	**1.000**	0.0	**0.0625**
test.c	31	**A_Read_2**	**A[i][j]**	936500	62500	**0.062**	0.0	**1.0**
test.c	31	**A_Write_7**	**A[i][j]**	999000	0	**0.0**	1.0	**1.0**

(a) Per-Reference Cache Statistics

Reference	Total Accesses	**Predictable Accesses**	**Regularity Ratio**	Avrg. Length	Distinct Strides	**% Stride Distribution**
B_Read_3	999000	999000	**1.0**	999	1	**stride=8000,100%**
A_Read_2	999000	999000	**1.0**	999	1	**stride=8,100%**
A_Write_7	999000	999000	**1.0**	999	1	**stride=8,100%**

(b) Per-Reference Stream Statistics

Figure 2.13: Original Per-Reference Memory Usage Statistics

This also causes the very low spatial reuse value, showing that less than 7% of the data cached by the `B_Read_3` reference is actually accessed by the processor before the memory line is evicted from cache. The stream metrics show that `B_Read_3` generated extremely predictable accesses (regularity ratio=1.0) with long stream lengths (average length=999) and only a single stride. The stride value is very large (8000), so no spatial locality is realized (since successive accesses map to different cache lines).

In contrast, `A_Read_2` has very good cache performance with excellent spatial reuse (100%). The stream metrics show that accesses generated by `A_Read_2` were also completely predictable (regularity ratio=1.0) with a long average stream length. In contrast to `B_Read_3` though, `A_Read_2` generated *single strided* accesses (of stride eight, the size of the `double` data type)). This ensured that `A_Read_2` achieved excellent spatial locality (spatial reuse=100%).

Upon closer inspection of the source code, we observe that the innermost loop (`j` loop) has a stride-1 traversal over the innermost dimension of the array A, which result in the accesses generated by the `A_Read_2` reference. In contrast, the accesses to array B are generated with the innermost j loop iterating over the *outermost* dimension of array B, leading to the high stride value (8000) seen for `B_Read_3`.

Optimization

The key idea is that both `A_Read_2` and `B_Read_3` generate completely predictable accesses. We exploit this fact to *prefetch* the array elements long before they are used to reduce the effective access latency. The average stream length for both access points is high, indicating that prefetching would be profitable, and the number of distinct strides is low, reducing the number of potential prefetch target addresses.

We evaluate this optimization on a Power4-based platform. This platform already has a hardware stream prefetcher that detects cache misses mapping to consecutive memory lines, such as frequently generated by stride-1 accesses. Once such a pattern is recognized, the prefetcher automatically prefetches the consecutive memory lines into cache [102]. Hence on this platform, there is no need to insert explicit prefetch instructions for the `A_Read_2` access point as it generates only stride-1 accesses. In contrast, accesses generated by `B_Read_3` will not be prefetched by the hardware prefetcher, since they do not map to consecutive memory lines (stride 8000). Hence, we target these accesses for prefetching.

We use the "Data cache block touch" (`dcbt`) prefetch instruction. The optimized code is as follows:

```
0 #define MATDIM 1000
1  double A[MATDIM][MATDIM], B[MATDIM][MATDIM];
2
3  void do_mult(void)
4  {
5     for(i=0;i < MATDIM;i++)
6         for(j=0;j < MATDIM;j++)
7            {
                  prefetch(&(B[j+15][i]));
8                 A[i][j] = A[i][j] * B[j][i];
9            }
10 }
```

The inserted instruction prefetches the B[][] element that will be accessed 15 iterations later (&B[j+15][i]). The number of iterations to "look-ahead" (15) is empirically chosen to ensure that the prefetch will complete before the prefetched data is accessed by the B[j][i] load instruction. Other values for the number of look-ahead iterations will still have a positive impact, as long as the prefetch is able to bring the memory line into the cache before the memory line is accessed.

Event	Original	Optimized	**% Improvement**
L1 Misses	1060733	62522	**94.10**
Processor Cycles	45325690	33013678	**27.16**

Figure 2.14: Performance of Original and Optimized Program

We used hardware performance counters to measure the number of L1 cache misses (event: `PM_LD_MISS_L1`) and the number of processor cycles (event: `PM_CYC`) for the original and the optimized program. The results are shown in Figure 2.14.

The prefetch instruction is very effective — it reduces the number of L1 cache misses by over 94%. This leads to a reduction in processor cycles of 27% over the original program.

We have shown how to use METRIC to select potential access points that can be targeted for prefetching. Even though the cache access pattern of `B` is statically determinable, none of the compilers we evaluated (IBM xlc 7.0, gcc 3.4) were able to generate prefetches targetting this access, even at very high optimization settings (xlc: -O5 -qprefetch -qtune=pwr4, gcc: -O3 -mpower). Thus, explicit prefetch insertion is still important in many cases to achieve good performance.

2.12.3 Use case: Detecting Conflict Misses

Consider the following snippet of C code:

```
23 double sumfunc(double S1[ ], double S2[], double S3[], int size)
24 {
25     int i;
26     double sum=0.0;
27
28     for(i=0;i < size;i++)
29      {
30            sum += S1[i] + S2[i] + S3[i];
31      }
32
33     return sum;
35 }
   #define MATDIM (8192)
   double A[MATDIM], B[MATDIM], C[MATDIM];
```

```
main(..)
 {
   ....
   result = sumfunc(A,B,C,MATDIM);
   ....
 }
```

The function `sumfunc` calculates the sum of the elements of the three arrays `A`, `B` and `C`. All these arrays contain elements of type `double` and have size `MATDIM`. This code was compiled into a program executable on the Power4 platform, using the IBM xlc compiler. The program executable was instrumented and the trace of memory accesses was obtained using our framework. The trace was used to simulate the operation of an L1 cache with the following parameters: size=128 KB, associativity=2, line size=128 bytes, writeback cache, LRU replacement policy. This configuration is similar to the L1 cache on the Power4 platform. For clarity, we ignore the other components of the memory hierarchy (L2 cache, DTLB) during the analysis of this example.

The overall performance of the cache was reported as:

hits	=	2	temporal hits	=	0
misses	=	24574	spatial hits	=	2
temporal ratio	=	0	spatial ratio	=	1.0
miss ratio	=	**0.99992**	**spatial reuse**	=	**0.06251**

This miss ratio is very high, almost all accesses were misses. The low `spatial reuse` value shows that, on average, only 6% of the memory line is used before it is evicted from the cache. These two indicators immediately point to the presence of a serious cache access inefficiency. The per-reference metrics are shown in Figure 2.15. Figure 2.15(a) shows the cache metrics generated by the simulator, while Figure 2.15(b) shows the per-reference stream metrics generated by the PRSD detector during trace compression.

Analysis

The per-reference results for all references shows very similar symptoms. All references almost always miss in cache and have low spatial reuse values. On the other hand, the stream metrics indicate that the references generated accesses that were highly predictable, with a regularity ratio of 1.0 and long average lengths (8192). Most crucially, each refer-

File	Line	Reference	SourceRef	Hits	Misses	Miss Ratio	Temporal Ratio	Spatial Reuse
test.c	30	**C_Read_2**	**C[i]**	0	8192	**1.0**	0.0	**0.0625**
test.c	30	**A_Read_0**	**A[i]**	1	8191	**1.0**	0.0	**0.0625**
test.c	30	**B_Read_1**	**B[i]**	1	8191	**1.0**	0.0	**0.0625**

(a) Per-Reference Cache Statistics

Reference	Total Accesses	**Predictable Accesses**	**Regularity Ratio**	Average Length	Distinct Strides	**% Stride Distribution**
C_Read_2	8192	8192	**1.0**	8192	1	**stride=8, 100%**
A_Read_0	8192	8192	**1.0**	8192	1	**stride=8, 100%**
B_Read_1	8192	8192	**1.0**	8192	1	**stride=8, 100%**

(b) Per-Reference Stream Statistics

Figure 2.15: Original Per-Reference Memory Usage Statistics

ence generated single-strided accesses (of stride eight, the size of the `double` data type), that normally would have led to extremely high spatial reuse values (since all elements in a cache line would be processed before the next memory line is fetched). Recall that for each reference, the simulator keeps track of the *evictor* reference, that removed data accessed by this reference from the cache. The list of evictors is shown graphically in Figure 2.16 and is the final piece of the puzzle. The arrows indicate the evictions — the head points to the reference that is evicted while the tail is the evictor. The edges are tagged with the percentage distribution of evictions, *i.e.*, the number of times this eviction occurred, among all evictions for a particular reference.

The evictor graph shows a clear cyclic pattern of evictors, with large eviction counts. The three references `A_Read_0`, `B_Read_1` and `C_Read_2` *conflict* in cache and evict each other's memory lines from the cache before the cache line could be fully used, which explains the low spatial reuse values.

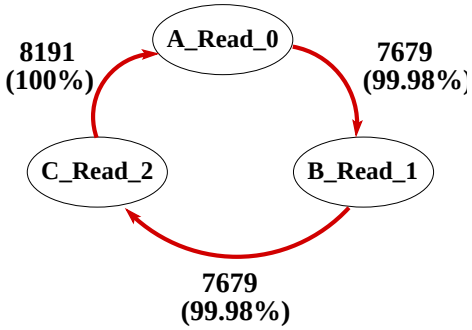


Figure 2.16: Evictor Graph

Optimization

We must update either the code or data layout to ensure that the references do not cause such a large number of conflict misses. We choose to remap the data layout by *padding* each data array with extra unused space. By padding, we hope to reduce the number of conflict misses, such that the spatial reuse inherent in the stride 1 accesses is exploited. In other words, we want to prevent evictions of data brought into the cache before all elements in the cache line have been accessed. The optimized code is shown below:

```
23 double sumfunc(double S1[ ], double S2[], double S3[], int size)
24 {
25     int i;
26     double sum=0.0;
27
28     for(i=0;i < size;i++)
29      {
30            sum += S1[i] + S2[i] + S3[i];
31      }
32
34     return sum;
35 }
    #define MATDIM (8192)
    double A[MATDIM+128], B[MATDIM+128], C[MATDIM+128];

    main(..)
    {
      ....
```

File	Line	Reference	SourceRef	Hits	Misses	Miss Ratio	Temporal Ratio	Spatial Reuse
test.c	30	**C_Read_2**	**C[i]**	**7680**	512	**0.062**	0.0	**0.986**
test.c	30	**A_Read_0**	**A[i]**	**7680**	512	**0.062**	0.0	**1.0**
test.c	30	**B_Read_1**	**B[i]**	**7680**	512	**0.062**	0.0	**1.0**

Figure 2.17: Optimized Per-Reference Memory Usage Statistics

```
  result = sumfunc(A,B,C,MATDIM);
  ....
}
```

Note the padding of the `A`, `B` and `C` arrays by 128 elements. This ensures that each iteration of the i loop maps to different cache sets for the `A[i]`, `B[i]` and `C[i]` accesses for the given cache configuration. In general, the padding could be parameterized so as not to be a multiple of the number of lines in associativity set. The updated code was compiled and run under our analysis framework as before. The following results were obtained:

hits	=	23037	temporal hits	=	0
misses	=	1539	spatial hits	=	23037
temporal ratio	=	0	spatial ratio	=	1.0
miss ratio	=	**0.0626**	**spatial reuse**	=	**0.99951**

Notice the significant decrease in the miss ratio and the dramatic increase in the spatial hits and spatial reuse value compared to the original program. The per-reference cache statistics are shown in Figure 2.17. The hits for all references have increased significantly and their spatial reuse approaches 1.0, the maximum possible value. Thus, we have successfully eliminated the large number of conflict misses in the original program. It is very hard for static compiler techniques to find such conflict misses, if not impossible in certain cases (*e.g.*, if arrays were passed as arguments at run time). Thus, we need tools like METRIC to analyze such scenarios.

2.13 Related Work

Compressed representations of memory traces have been commonly represented in past work, both in hardware and software. For example Havlak *et al.* use regular section descriptors (RSDs) for compiler analysis. Weikle *et al.* describe a trace specification notification called TSpec that can encode information similar to an RSD [109]. However, TSpec is more complex because it is directly analyzed for cache evaluation by an analytical component called a cache "filter". In contrast, we decompress RSDs on-the-fly and use the generated accesses for incremental cache simulation.

There are a number of binary rewriting tools presented in past work. Atom is a static binary rewriting tool that inserts instrumentation into application binaries [99]. Dynamic binary rewriting is an enhancement that allows selective instrumentation (both location and time-wise) of executing applications. This is useful for periodic monitoring of long-running programs. Examples of dynamic binary rewriting include DynInst([10]) and PIN([87, 59]).

Additional hardware support for determining cache miss causes have been proposed (*e.g.*, informing memory operations) but these are not supported in contemporary processors [42, 72].

Several tools provide aggregate metrics obtained at low cost from hardware performance counters. HPCToolkit uses statistical sampling of performance counter data and allows information to be correlated to the program source [70]. A number of commercial tools (*e.g.*, Intel's VTune, SGI's Speedshop, Sun's Workshop) also use statistical sampling with source correlation, albeit at a coarser level than HPCToolkit or our approach. Hardware counters are usually limited in number and typically have restrictions on the type of events that can be counted simultaneously. Hardware counters complement our methodology. Aggregate metrics provided by these counters can be used to determine whether a cache bottleneck exists, and then our tool can be used to generate detailed source-tagged statistics to isolate and understand the bottleneck.

Recent work by Mellor-Crummey *et al.* uses source to source translation on HPF to insert instrumentation code that extracts a data trace of array references. The trace is later exposed to a cache simulator before miss correlations are reported [70]. This approach shares its goal of cache correlation with our work. CProf [55] is a similar tool that relies on post link time binary editing through EEL [53, 54] but cannot handle shared

library instrumentation or partial traces. Lebeck and Wood also applied binary editing to substitute instructions that reference data in memory with function calls to simulate caches on-the-fly [56]. Our work differs 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. Another major difference addresses the overhead of large data traces inherent to all these approaches. We restrict ourselves to partial traces, employ trace compression to provide compact representations and derive stream metrics indicating cache bottlenecks during compression.

Recent work by Chilimbi *et al.* concentrates on language support and data layout to better exploit caches [24, 23] as well as quantitative metrics to assess memory bottlenecks within the data reference stream [21]. This work introduces the term *whole program stream* (WPS) to refer to the data reference stream, and presents methods to represent the WPS compactly in a grammatical form. However, their work focuses on prefetching for dynamically allocated data while we focus on reference reordering through code transformations to improve data locality. Furthermore, our compression algorithm for reference streams caters to regular array accesses with lower complexity that a WPS with its need for states and transitions. Ding and Zhong *et al.* predict program locality from profiles using the approximate reuse distance of referenced data to identify regular and irregular reference patterns [31]. Their work is continued by Zhong *et al.* in analyzing the hierarchical relation between program data and modeling it very effectively with k-distance analysis, which provides the means to identify beneficial data layout transformations [116]. Our method, in contrast, provides per-reference cache information that indicates benefits for code transformations by pinpointing references participating in cache evictions.

Other efforts concentrate on access modeling based on whole program traces using cache miss equations [34] or symbolic reference analysis at the source level based on Presburger formulas [19]. These approaches involve linear solvers with response times on the order of several minutes up to over an hour. The feasibility of using these approaches has not been demonstrated on large programs, but only with small kernels like matrix multiply.

A number of approaches address dynamic optimizations through binary translation and just-in-time compilation techniques for native code [95, 5, 25, 105, 36]. The main thrust of these techniques is program transformation based on knowledge about taken execution paths, such as trace scheduling. The transformations include the reallocation of registers and loop transformations (such as code motion and unrolling), to name a few. These efforts

are constrained by the tradeoff between the overhead of just-in-time compilation and the potential payoff in execution time savings. Our approach differs considerably. We allow offline optimizations to occur, which do not affect the application's performance during compilation, and we rely on injection of dynamically optimized code thereafter.

SIGMA is a tool using binary rewriting through Augmint6k to analyze memory effects [30]. This is the closest related work. SIGMA captures full address traces through binary rewriting. Experimental results show a good correlation to hardware counters for cache metrics of entire program executions. Performance prediction and tuning results are also reported (subject to manual padding of data structures in a second compilation pass in response to cache analysis). Our approach differs in several respects. First, our cache analysis is more powerful. In addition to generating per-reference cache metrics, we also generate per-reference *evictor* information. We supplement these results with *stream* characteristics observed by the compression algorithm at each access point. This allows us to infer potential for more sophisticated transformations, as demonstrated by the examples in the preceding sections. Second, their work lacks an evaluation of the efficiency and overhead of the compression algorithm used. In contrast, we demonstrate that our trace compression algorithm compresses better than the state-of-the-art in trace compression for 7 out of the 12 benchmarks we evaluated, and has comparable performance on the rest. Finally, our framework is designed for collecting and processing *partial* access traces. In contrast, their work neither captures partial traces nor presents a concept for such an approach.

In our previous work, we used binary rewriting to extract the memory access stream and characterize its *spatial* regularity [71]. In that work, we used regularity values to classify applications as regular or irregular and showed how particular regularity metrics suggested specific applicable optimizations (*e.g.*, long length regular streams are amenable to prefetching). Our current work differs in many respects. In this work, we segregate the memory access stream by *access point* and calculate regularity metrics for each point separately. In contrast, our previous work calculated a single regularity value for the entire program or program segment. Here, we provide more fine-grained information on the memory access behavior. More importantly, our current work supplements the stream metrics with cache usage metrics (per-reference statistics, evictor information). The richer information about the potential memory access inefficiencies enables the use of more sophisticated optimizations.

Our recent work beyond uniprocessor METRIC describes a binary rewriting based

framework to characterize shared memory coherence metrics for OpenMP programs [66]. In that work, we use software instrumentation to extract synchronization information and memory access traces for each OpenMP thread, and use these for incremental coherence simulation. Even more recently, we extended this approach to investigate the benefits from hardware support to gather "lossy traces" that are then utilized to analyze coherence traffic [67]. Our work, in contrast, concentrates on application level characterization of *uniprocessor* memory hierarchy metrics.

2.14 Conclusion

In this work we demonstrate that dynamic binary rewriting offers novel opportunities for detecting inefficiencies in memory reference patterns. Our contributions are a framework to instrument selective load and store instructions on-the-fly for generating partial access traces, a novel trace compression algorithm for compressing these traces and a cache simulation framework that generates detailed source reference tagged metrics. We evaluated our compression algorithm with respect to compression rate and overhead. We demonstrated that the compression rate is better than the state-of-the-art for the majority of the benchmarks (7 out of 12), and comparable for the rest.

Our framework generates a rich set of performance metrics describing the memory access behavior of the program, including per-reference cache metrics, evictor information and stream metrics generated by the compression algorithm. We demonstrated how these metrics enable the detection and understanding of memory access inefficiencies with several use cases. METRIC has a potential advantage over compile-time analysis when analyzing these performance inefficiencies for sample codes, particularly if interprocedural analysis is required.

Chapter 3

Source-Code Correlated Cache Coherence Characterization of OpenMP Benchmarks

3.1 Summary

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.

3.2 Introduction

High-performance computing platforms are increasingly deployed in configurations of multiprocessor shared-memory nodes. Understanding the coherence behavior of multithreaded 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.*, [7, 14, 28, 43, 80, 88]), 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.*, [52, 93]).

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 work 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 [107, 108].

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 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.

3.3 ccSIM Framework

Figure 3.1 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

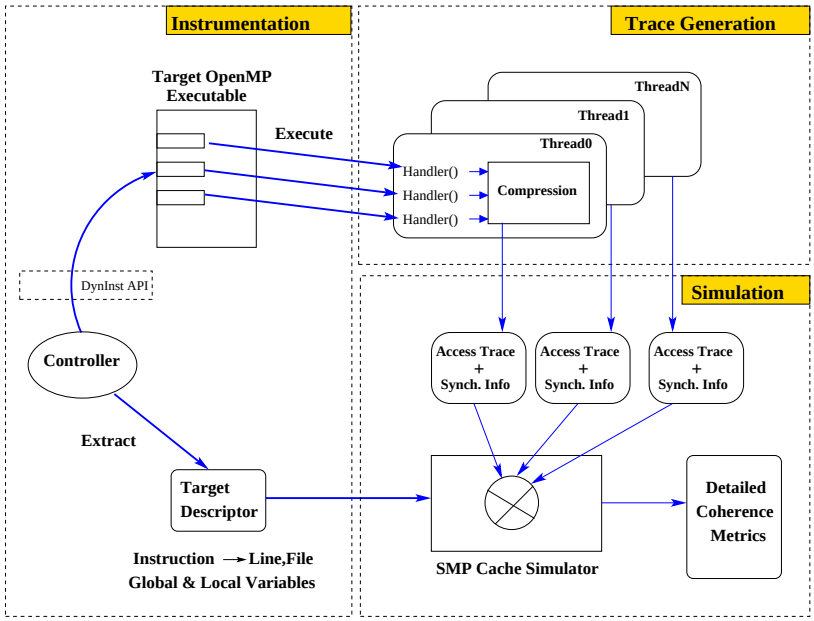


Figure 3.1: ccSIM Framework

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.

3.3.1 Instrumentation

Our instrumentation tool uses the DynInst instrumentation library [10] for dynamic program instrumentation. It is an extension of our previous work in using binary rewriting to extract memory traces from uniprocessor programs [65]. 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.

3.3.2 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

[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.

accesses, function entry/exits and OpenMP synchronization 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 work, 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.

3.3.3 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.

3.3.4 Studying Invalidations and Misses

For optimizing the sharing behavior of multi-threaded applications, identifying the sources of invalidations and coherence misses is essential. Invalidations occur when a processor writes to a shared cache line. Coherence misses occur when a processor accesses a recently invalidated cache line. Coherence misses are very expensive because the concerned memory line will be absent from *all* levels of cache and will have to be brought in from the other processor's cache over the bus. The latency for this operation is high enough that contemporary out-of-order processors will run out of independent instructions and will be forced to stall. Thus, reducing the number of coherence misses often has a significant impact on the overall wallclock execution time of the program.

[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 [113].

By reducing invalidations we will also reduce the number of coherence misses. Given the importance of invalidations, we provide multiple sub-classifications for different types of invalidations.

First, we are able to distinguish between *true* and *false* sharing invalidations by keeping information about parts of the cache line that are accessed by the host processor. True-sharing invalidations occur when multiple processors access a particular data element and at least one of the accesses is a write. False-sharing invalidations occur due to accesses to *different* data elements that happen to be resident in the same memory line. False-sharing invalidations are an artifact of the limitation that coherence information is only maintained at cache line level granularity in contemporary architectures. Consider two un-related data items d1 and d2 that are resident in the same memory line, and the line is in shared state in the caches of processor P1 and P2. If P1 writes to d1, the memory line containing d1 and d2 is invalidated from P2's cache. If P2 subsequently attempts to access d2, it will experience a coherence miss.

Second, we distinguish between *in-region* and *across-region* invalidations. We introduced the concept of a *region* above (Section 3.3.3). Within the same region, we further distinguish true-sharing invalidations by whether they are protected by a critical region or not.

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 [65]. 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 [66] 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 later in this chapter.

3.4 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 [48] plus an additional OpenMP benchmark (NBF) from the GROMOS benchmark suite [37].

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 [1], and one benchmark (AMMP) from the SPEC2001M OpenMP suite [98]. More details about these applications are presented in the case studies.

3.4.1 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

Table 3.1: 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 3.2: HPM vs. ccSIM

Benchmark	HPM	ccSIM Interleaved	ccSIM Piped	% Error 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

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 3.1 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 3.2 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 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 3.3.4. The cause of the discrepancy becomes apparent when we examine the in-region true-sharing critical invalidations shown in Figure 3.2. 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]

[4]In Figure 3.2, 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

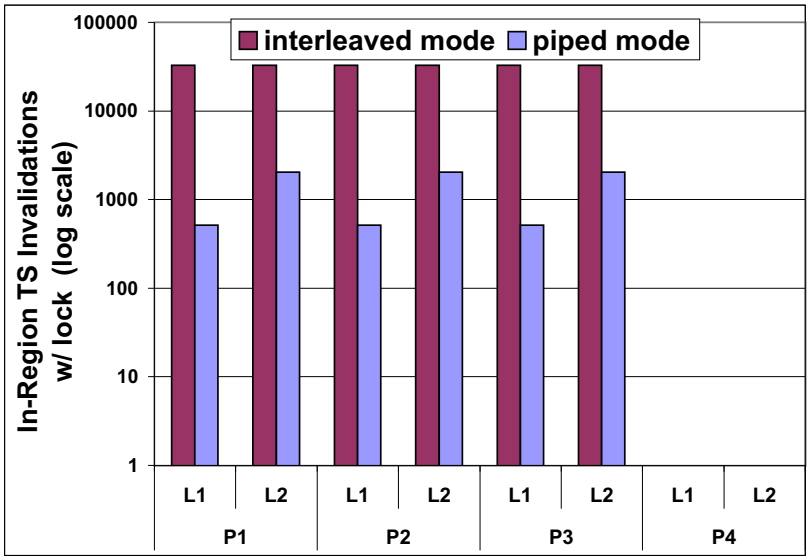
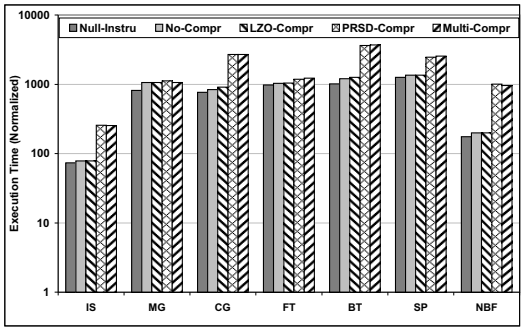
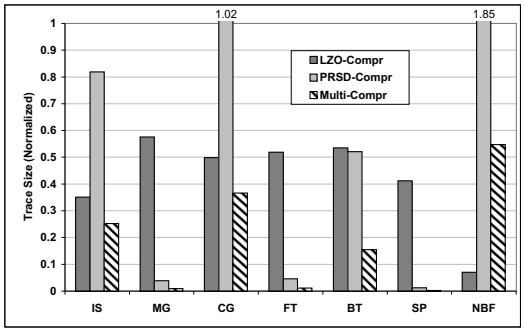


Figure 3.2: NBF: Interleaved and Piped



(a) Execution Time (Normalized)



(b) Trace Size (Normalized)

Figure 3.3: Execution Overhead and Trace Sizes

3.4.2 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.

For compression, we compare the following strategies: 1) **No Compression (No-**

other processors.

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 [62]. 3) **LZO Compression (LZO-Compr)**: This is an open-source lossless compression library designed specifically for compression speed [85]. 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 3.3(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 overhead. For some benchmarks (MG, FT, SP) the overhead is low while for others, there is significant overhead compared to LZO compression.

Trace Compression: Figure 3.3(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.

3.5 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 [67]). 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.

3.5.1 NBF: Non-Bonded Force Kernel

This case study is also discussed in Anita Nagarajan's thesis [75]. We included it in this document in order to maintain the readability of the text and to provide additional examples of ccSIM in action.

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 3.4 shows the breakdown of misses for L1 and L2 caches for each processor obtained by ccSIM. 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 3.4 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.

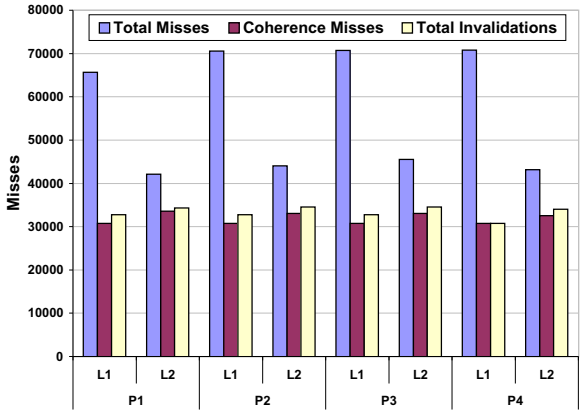


Figure 3.4: NBF: Breakdown of L2 misses

Table 3.3: 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

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 3.3 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

Table 3.4: 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 3.5: NBF: L2 Invalidations (HPM raw)

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

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)
        barrier();
   }
```

Results: Table 3.3 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 3.4 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 3.5 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 3.5). Second, the restructured programs have lower OpenMP execution overhead because they execute fewer OpenMP calls.

3.5.2 IRS: Implicit Radiation Solver

IRS-1.4 is part of the ASCI Purple codes [1]. 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`).

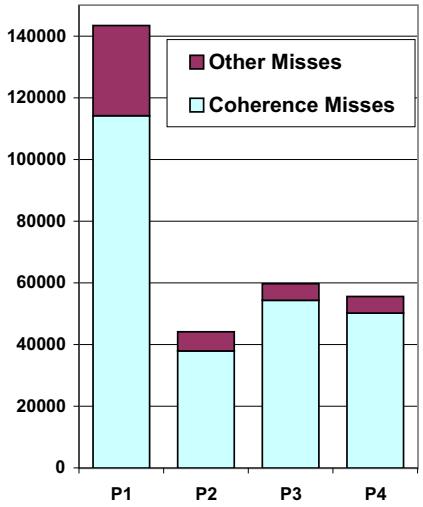


Figure 3.5: IRS: Breakdown of L2 misses

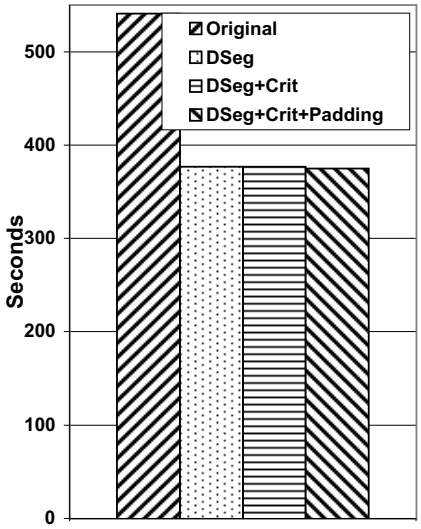


Figure 3.6: IRS: Time w/ 4 Optimizations

Analysis: Figure 3.5 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 3.7 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; these are represented by a

Figure 3.7: IRS: Per-Reference Statistics

Proc	No.	Reference	Grp	Coh. misses	Invalidations True in	Invalidations True across	Invalidations False in	Invalidations False across	Optimization strategy	Opt. Coh. 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 */
```

Figure 3.8: IRS Breakup into Parallel Regions

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 3.8.

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 3.7. 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 3.6 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.

3.5.3 SMG2000: Semi-coarsening Grid Solver

Table 3.6: SMG: Per-Reference Statistics (Processor-1)

No.	Reference	Group	Coherence Misses	Invalidations				Optimization Strategy	Optimized Coherence Misses
				True		False			
				In	Across	In	Across		
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**

SMG2000 is part of the ASCI Purple benchmark set [1]. The SMG code utilizes the `hypre` library [32], that can 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

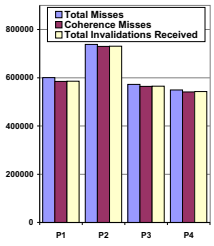


Figure 3.9: Breakdown of L2 misses

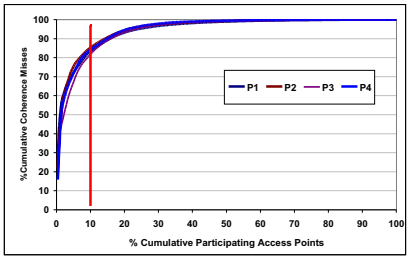


Figure 3.10: SMG: Cumulative L2 Coherence Misses

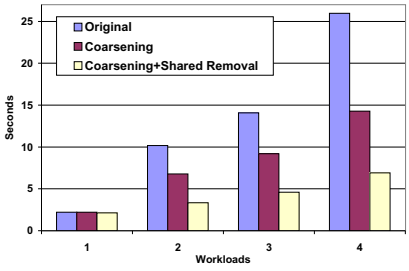


Figure 3.11: SMG: Time for different Workloads

for different optimization strategies.

Analysis: For all processors, the L2 miss rate is quite high, ranging from 64% to 81%. Figure 3.9 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 3.10 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 3.6. 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 3.6. 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 3.11 compares the wall-clock times for the different workloads.

`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).

3.5.4 AMMP: Molecular Mechanics Program

AMMP is a part of the SPEC2001M OpenMP benchmark suite [98]. We use the smaller `test` data set for characterizing the coherence behavior of the benchmark, and later

Table 3.7: AMMP: Per-Reference Statistics

#	Reference			Group	Coh. misses	Invalidations				Optimization strategy
						True		False		
	file	line	name			in	across	in	across	
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**	

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 3.7 shows the top references exhibiting coherence misses for processor 3. The results for other processors are similar.

Invalidator Lists: Figure 3.12 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

[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.

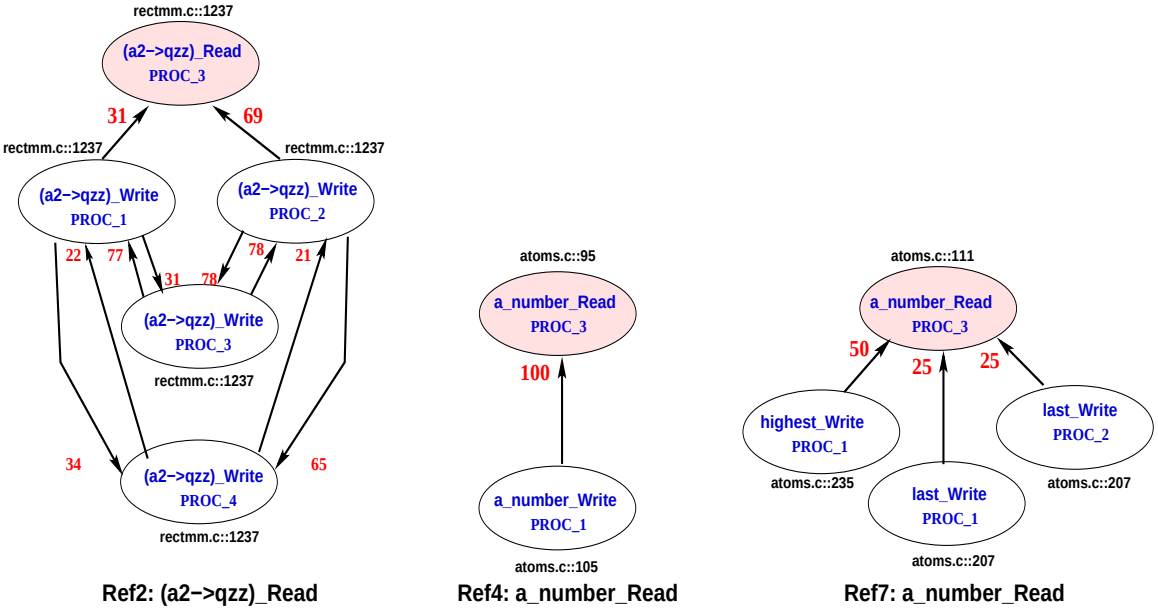
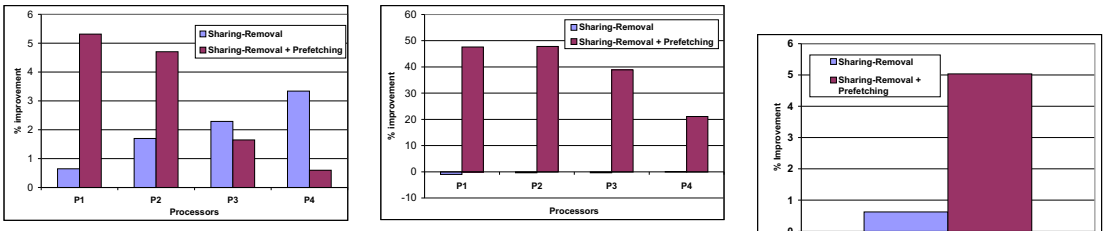


Figure 3.12: Invalidators for Selected References

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 3.12. 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: As before, 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.



(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

Figure 3.13: Reduction in Execution Metrics for AMMP

The invalidator lists for reference `(a2->qzz)_Read` are shown in Figure 3.12. 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 pre-load 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 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 3.12 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 3.13. 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 3.13(a) and 3.13(b) shows the reduction in per-processor cycles and per-processor L1 data cache misses, over the original version. Figure 3.13(c) shows the reduction 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 ccNUMA systems where the cost of accessing remote memory and remote caches is higher than the cost of accessing their local counterparts [64].

`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.

3.5.5 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 [1], 301.wupwise_m from the SPEC OMP2001M suite and FT from the C OpenMP version of the NAS-2.3 suite [2]. 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 3.14. 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 inval-

```
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 + (....)*ndata*nbdy2
         ........
1083:  mm4 = mm4 + (...)*nbdy*nbdy2
         ........
1090:  mm5 = mm5 + (...)*nbdy2*nbdy2
1092:  >>)
```

Figure 3.14: 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
```

Figure 3.15: 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: }
```

Figure 3.16: FT, compute_indexmap in ft.c

idations. 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 3.15. 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 3.16 (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 3.5.3). 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.

3.6 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 [35]. At one end of the spectrum are complete software machine simulators. RSim is a simulator for ILP multiprocessors with support for CC-NUMA architectures with a invalidation-based directory mapped coherence protocol [43]. SimOS is a complete machine simulator capable of booting commercial operating systems [88]. 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 [80], Proteus [7] and Tango [28] 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 [69] and CProf [55] are cache profilers that aim at detecting memory bottlenecks. CProf relies on post link-time binary editing through EEL [53, 54]. Lebeck and Wood also applied binary editing to substitute instructions that reference data in memory with function calls to simulate caches on-the-fly [56]. Other approaches rely on hardware support, such as watchdogs [11] or statistical sampling with hardware support in ProfileMe [29], to gather information on data references. Scal-Tool detects and quantifies scalability bottlenecks in distributed shared memory architectures, such as the SGI Origin 2000 [97]. 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 [82].

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 [8]. 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" [8]. 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) [30] 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 [68]. 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 [9]. 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. HPC-Toolkit uses statistical sampling of performance counter data and allows information to be correlated to the program source [70]. 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 [35]. 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 [52]. 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 [93]. 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.

3.7 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 online 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.

Chapter 4

Analysis of Cache Coherence Bottlenecks with Hybrid Hardware/Software Techniques

4.1 Summary

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.

4.2 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.

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 [66]. We showed that code transformations can result in up to 73% improvement in wall-clock time for large-scale NNSA ASC benchmarks [1], 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 work 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 [14, 43, 80, 7, 28] to the operating system interface [88]. Past work on the performance tuning concentrates on program analysis to derive optimized code [52, 93].

More recent work on identifying coherence bottlenecks is based on tracing memory accesses *via* dynamic binary rewriting [66, 101]. 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 work 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 4.5.1 details the metrics,

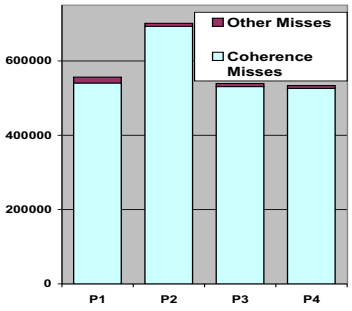
namely coverage fraction (most frequent coherence misses we detect *vs.* those in the full trace) and number of false positives (references that we false identify as coherence misses).

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 4.5.1, 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 work 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.



(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

Figure 4.1: Characterization for SMG2000

The rest of the chapter 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.

4.3 Source-Correlated Statistics

In prior work, we used a full-tracing approach to extract complete access traces from OpenMP applications [66]. These traces were fed to an incremental coherence simulator, which generated detailed source-code correlated coherence metric information. In this work 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 [1]. The example stems from our prior work [66]. 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 4.1(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 4.1(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.

4.4 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 [66, 71].

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 4.5.1 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.

4.4.1 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.

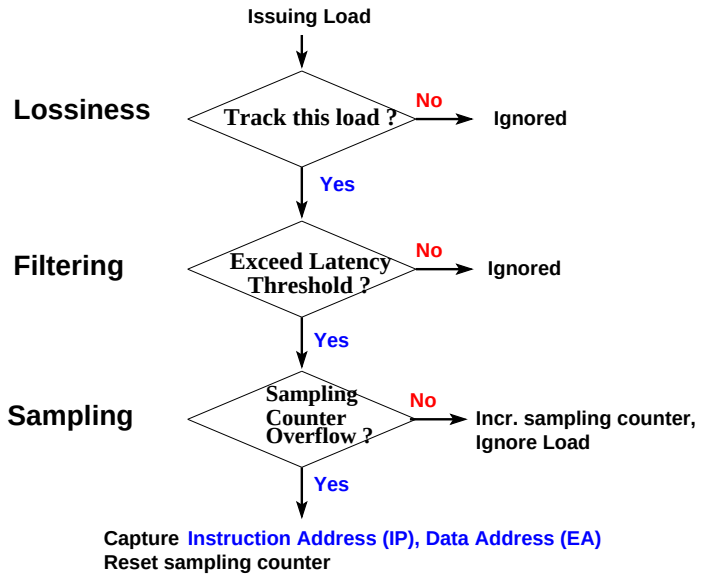


Figure 4.2: Simplified PMU Operation

Hardware-based support for obtaining data traces is beginning to be available on a few high-performance architectures (*e.g.*, on the Itanium-2 and Power architectures). The most sophisticated and flexible, yet readily accessible support at the user level is found on the Itanium-2 [47].

PMU Operation

A simplified view of the Itanium-2 Performance Monitoring Unit (PMU) operation for tracing long-latency loads is shown in Figure 4.2; full details are available elsewhere [46]. 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 [46].

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.

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 4.4.2) 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 4.8. 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 work 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.

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 partic-

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

ular load instruction as it is dispatched into the pipeline. This reduces the likelihood that consecutive tracking candidates originate from the same load (IP) and, thus, spreads the tracked loads over multiple references (IPs) in tight loops.

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

4.4.2 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 work). 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]

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.

[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.

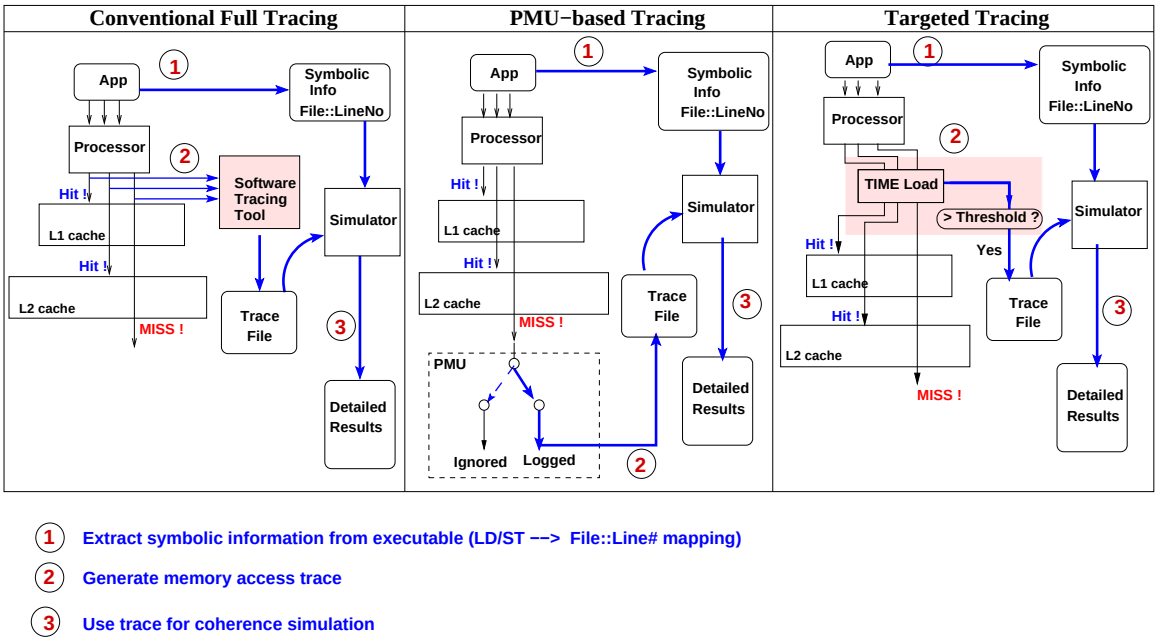


Figure 4.3: Comparison of Trace-Based Methods

4.5 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 4.3 shows a high-level comparison of the full-trace based method and the two hardware-assisted methods that we introduce in this work. In all of our methods, a memory access trace is generated for the target benchmark. The trace is used offline for incremental coherence simulation. The coherence simulator associates coherence metrics with high-level source code constructs using symbolic information extracted from the target executable. The chief difference between the methods lies in the generation of the memory access trace. The naïve software tracer (extreme left in Figure 4.3) 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 [87, 59]. 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 [65, 66] using DynInst [10]. 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 work we model the MESI coherence protocol that our target platform uses [46]. 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

Table 4.1: 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

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 4.1.

The NAS benchmarks are C language OpenMP versions of the original NAS-2.3 serial benchmarks [4] provided by the Omni Compiler group [2]. `SMG2000` and `sPPM` are part of the ASCI Purple benchmark set [1]. The benchmarks are used with comparatively small data sets (class S for NAS) since the 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.

4.5.1 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 work, 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 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

[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 $\frac{X}{Y}$.

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.

4.6 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 [46]:

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

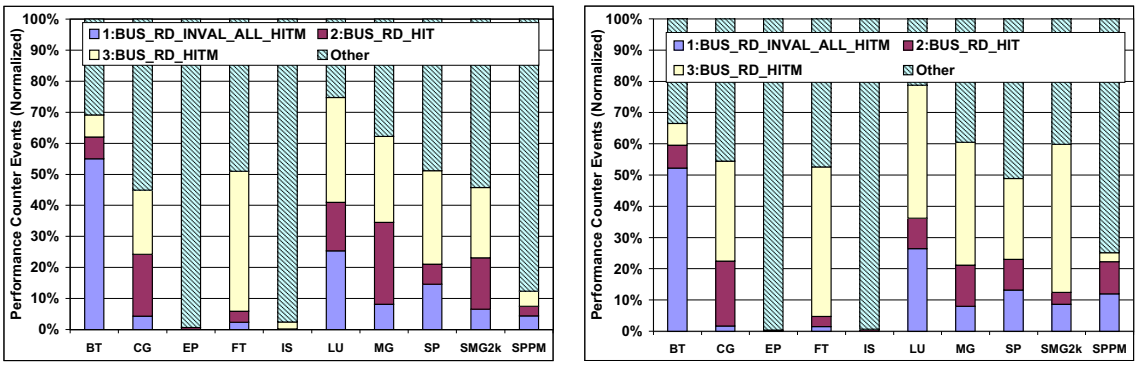
Event 3, BUS_RD_HITM Bus Read Hit Modified Non-local Cache Transactions

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.



(a) Processor-1

(b) Processor-2

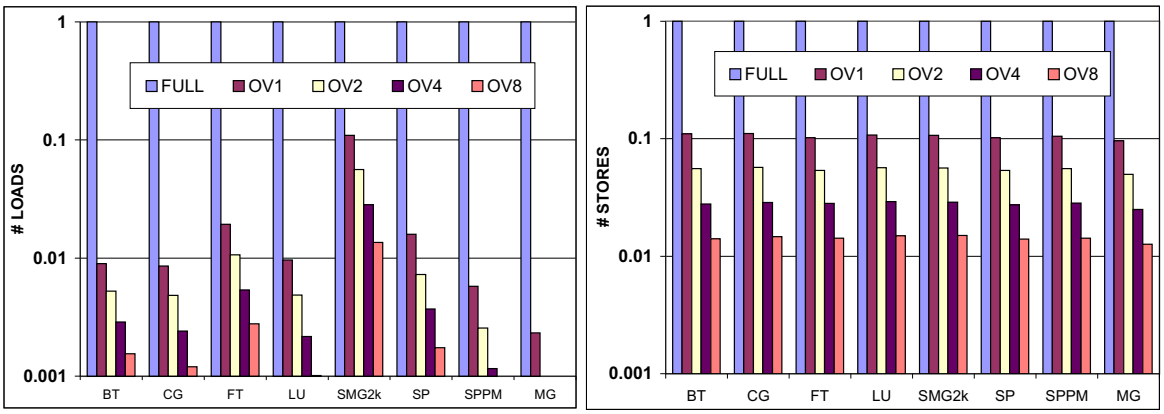
Figure 4.4: Characterization using Hardware Performance Counters

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 4.4(a) and 4.4(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 provides detailed source code-correlated statistics that provide a "drill down" into the bulk



(a) Loads Traced

(b) Stores Traced

Figure 4.5: Memory Accesses Traced with PMU-based tracing, Normalized to Number of Accesses in Full Trace

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.

4.7 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 (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 [46]. 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]

4.7.1 Trace Sizes

Figures 4.5(a) and 4.5(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.

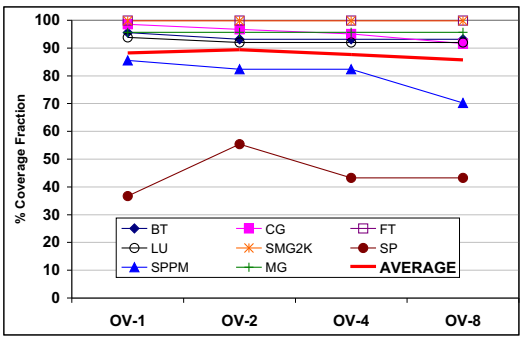
The graphs show that our method decreases the number of accesses collected by one to two orders of magnitude compared to full tracing. This considerable decrease results from the PMU's ability to discriminate and track only long-latency loads, ignoring the far more frequent low-latency accesses that 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.

4.7.2 Accuracy of Results

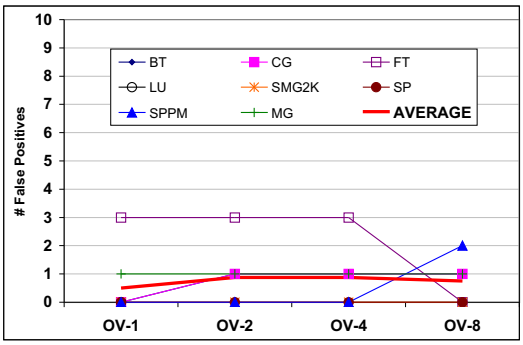
Figures 4.6 and 4.7 depict the accuracy of the results using lossy traces for the two metrics of *invalidations caused* and *coherence misses*. Due to space constraints, only the results for processor 1 are shown. The results for the other processor are similar.

We compare the accuracy of the results using the yardsticks of *coverage fraction* and *number of false positives*, as described in section 4.5.1.

[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.



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



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

Figure 4.6: Top-10 References Causing Invalidations on Processor 1, PMU Sampling Rates of 1-8

Metric: Invalidations Caused

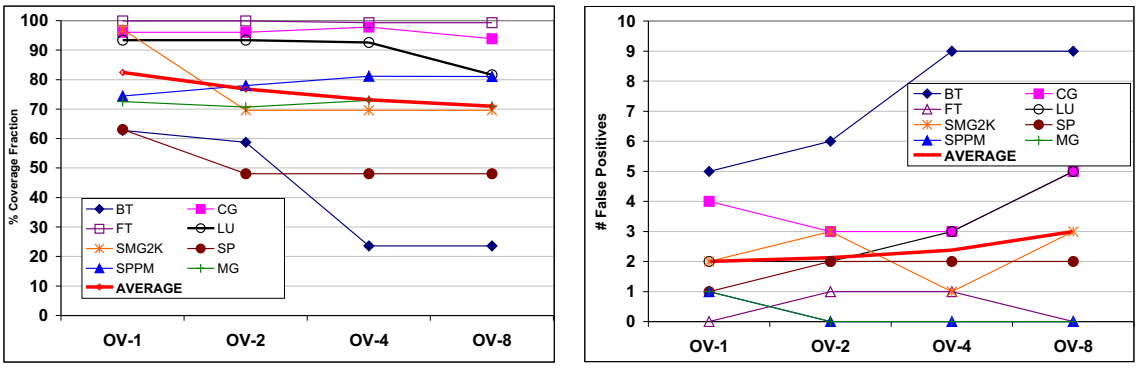
Consider the results for *invalidations* depicted for coverage fraction and the number of false positives in Figures 4.6(a) and 4.6(b), respectively. The results are shown for different sampling intervals (OV1 to OV8). For OV1, the coverage fraction ranges from 36-100%, averaging 86%. Except for SP, 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



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

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

Figure 4.7: Top-10 References Resulting in Coherence Misses on Processor 1, PMU Sampling Intervals of 1-8, PMU-based tracing

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).

Metric: Coherence Misses

Figures 4.7(a) and 4.7(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 4.7(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.

4.7.3 BT

As seen in the last section, lossy-trace based simulation generates very poor coherence miss results for BT. As Figures 4.7(a) and 4.7(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 4.4. 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 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.

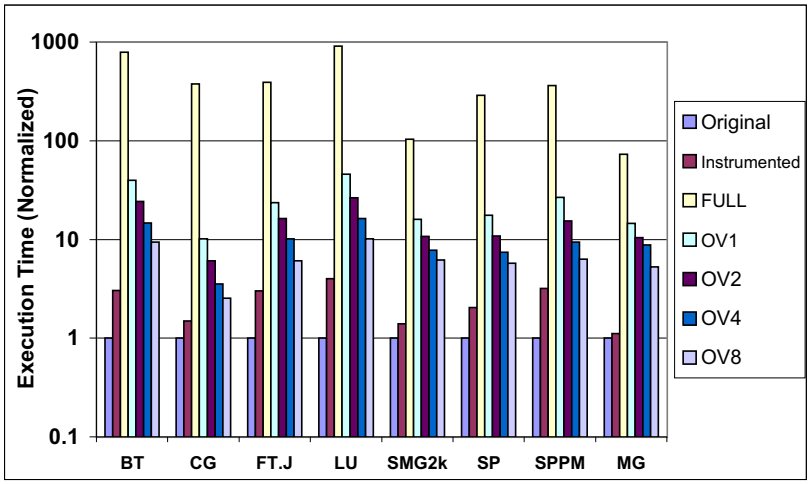


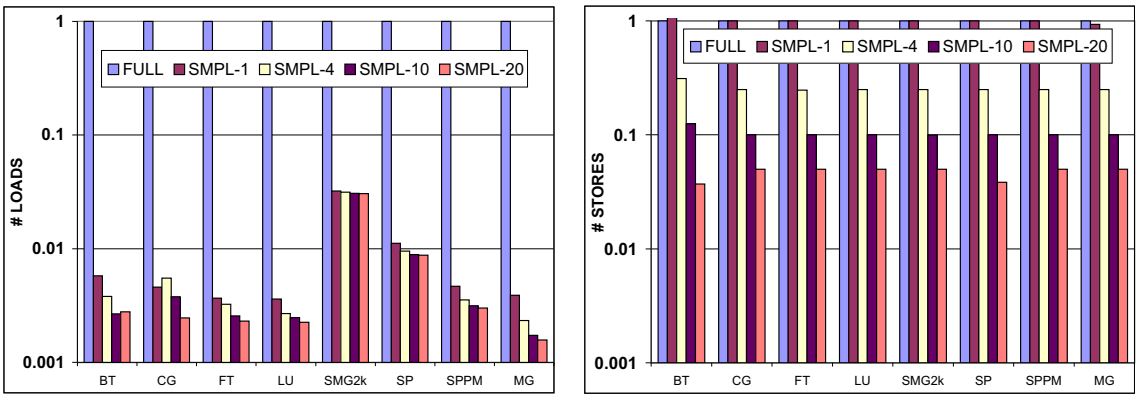
Figure 4.8: Execution Time: Full-tracing *vs.* PMU-based tracing

4.7.4 Execution Overhead

Figure 4.8 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 4.4 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.

4.8 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,



(a) Loads Traced

(b) Stores Traced

Figure 4.9: Memory Accesses Traced with Targeted tracing, Normalized to Number of Accesses in Full Trace

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 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).

4.8.1 Trace Sizes

Figures 4.9(a) and 4.9(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 4.9(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]

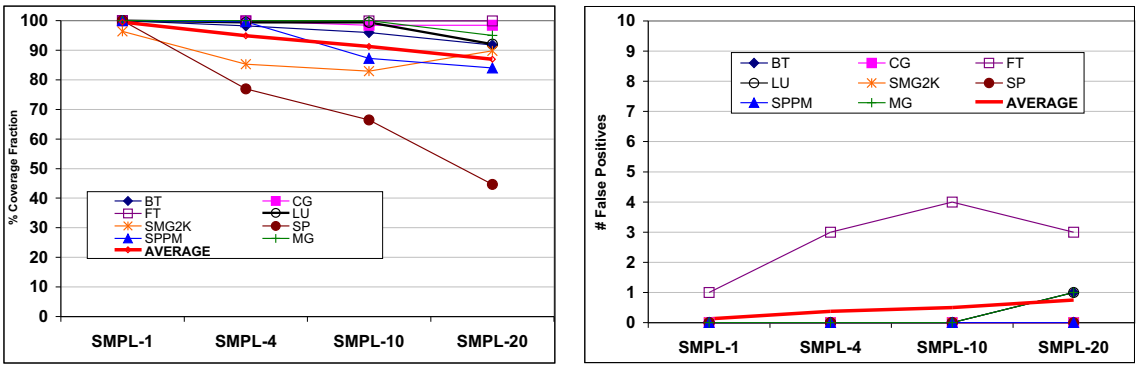
Figure 4.9(b) shows that the number of stores captured decreases as expected when the software sampling interval is increased from SMPL-1 to SMPL-20. The impact of this lossy tracing of stores is explored in the following paragraphs.

It is instructive to compare these figures *vs.* the corresponding ones (Figures 4.5(a) and 4.5(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 differing 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.

4.8.2 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 4.10 and 4.11 show these yardsticks applied to the metrics of *invalidations caused* and *coherence misses*, respectively. The results shown are for processor-1.

[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

Figure 4.10: Top-10 References Causing Invalidations on Processor 1, Store Sampling Rates of 1,4,10,20

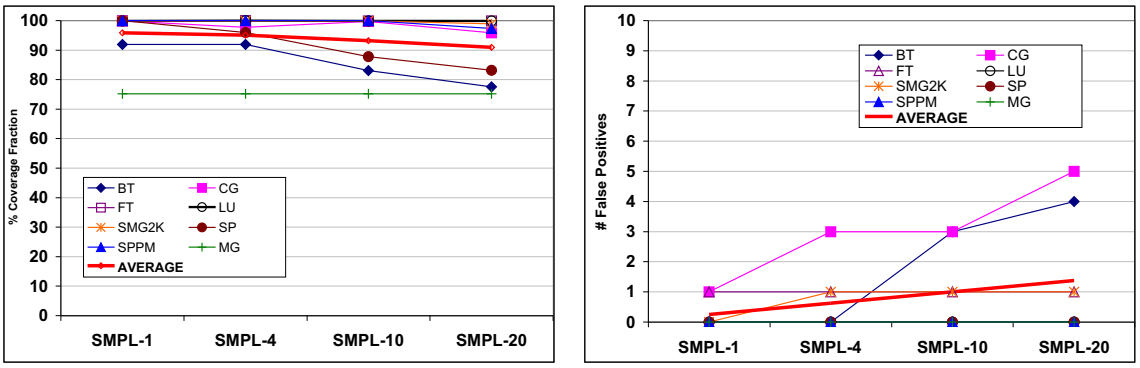
Metric: Invalidations Caused

Figures 4.10(a) and 4.10(b) show the coverage fraction and number of false positives for this metric, over different store sampling intervals 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 4.6(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 4.7.2). These false positives again appear because of lossiness in the store trace that is captured (Section 4.7.2) at sampling intervals other than SMPL-1.

Metric: Coherence Misses

Figures 4.11(a) and 4.11(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 for Targeted Tracing vs. Full-Tracing

(b) Number of False Positives with Targeted Tracing Approach

Figure 4.11: Top-10 References Resulting in Coherence Misses on Processor 1, Store Sampling Rates of 1, 4, 10, 20

a coverage fraction greater than 91%, except for MG (75%). The average number of false positives is less than one (Figure 4.11(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 4.6(b)). Thus, these benchmarks are very sensitive to the degree of lossiness of stores.

4.8.3 Execution Time

Figure 4.12 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 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.

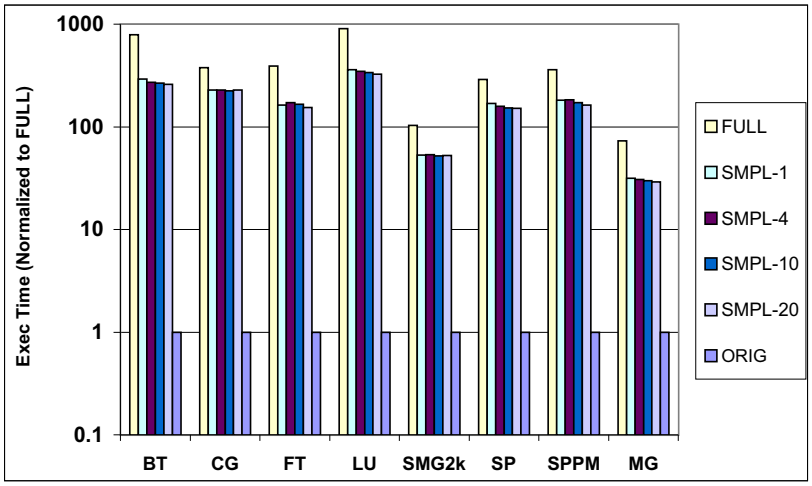


Figure 4.12: Execution Time: Full-tracing *vs.* Targeted tracing

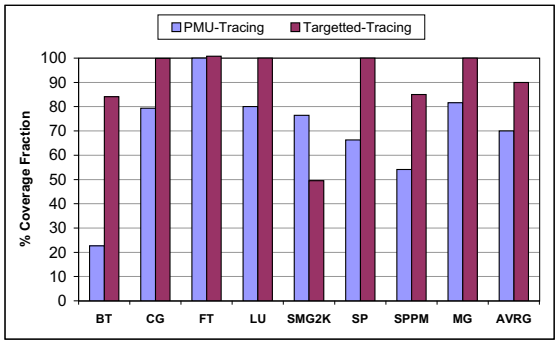
The execution time savings, though useful, are not as dramatic as those for PMU-based tracing. The chief benefit of targeted tracing over full tracing comes in the large trace size reductions (especially loads), which makes it easier and faster to use the trace for offline activities, *e.g.*, for incremental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

4.9 Comparing True Sharing and False Sharing

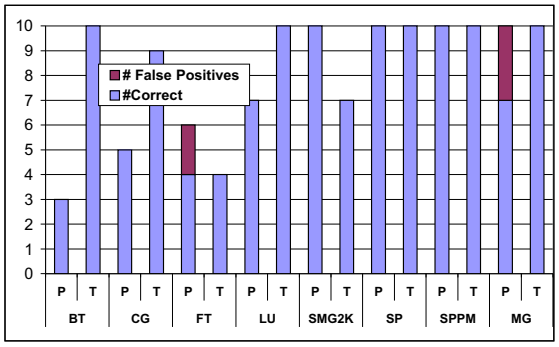
We maintain per-cache-line bit vectors indicating which parts of the cache line have been accessed (either by a load or a store) from the attached processor. This allows us to classify stores that cause invalidations as either *true-sharing invalidations* or *false-sharing invalidations*. When a store 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 [66].

In the results below, the PMU-based runs used a sampling interval of 1 (OV-1).



(a) Coverage Fraction



(b) Correct References vs. False Positives

(P=PMU-based Tracing, T=Targeted Tracing)

Figure 4.13: Coverage and False Positives for PMU-based and Targeted Tracing with Respect to True-sharing Invalidations

The targeted tracing used store sampling of 1 (SMPL-1), *i.e.*, all stores are captured.

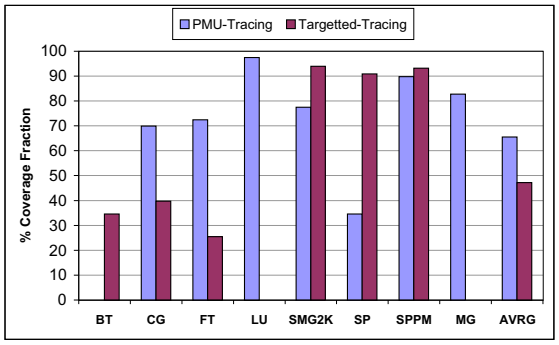
4.9.1 Comparing True Sharing Invalidations-Caused

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.

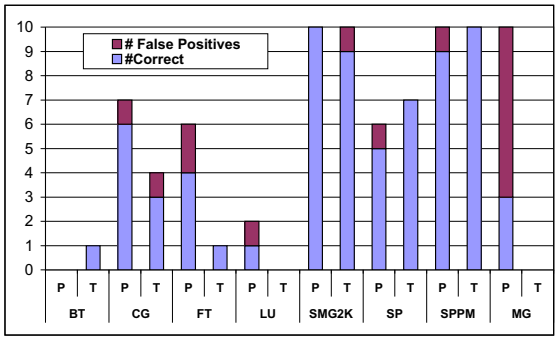
Figures 4.13(a) and 4.13(b) compare the coverage fraction and number of false positives.

We observe:

- Consider Figure 4.13(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



(a) Coverage Fraction



(b) Correct References vs. False Positives

(P=PMU Tracing, T=Targeted Tracing)

Figure 4.14: Coverage and False Positives for PMU-based and Targeted Tracing with Respect to False-Sharing Invalidations

Table 4.2: 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**

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.

4.9.2 Comparing FalseSharing Invalidations-Caused

The benchmarks we considered had distinctly different false-sharing behavior. Table 4.2 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* [110]) 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 4.14(a) and 4.14(b) show the coverage fraction and number of false posi-

tives 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 4.2).

- Consider MG: PMU-based tracing produces many false positives. Table 4.2 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 when there inherently exists no false sharing in the benchmark, though that occurred for only one benchmark in our evaluation.

4.9.3 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

Table 4.3: 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:		Similar ?
		PMU-Tracing	Full Tracing	
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**

benchmark. Table 4.3 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 4.2. 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 4.9.2). 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 (*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

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

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.

4.10 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

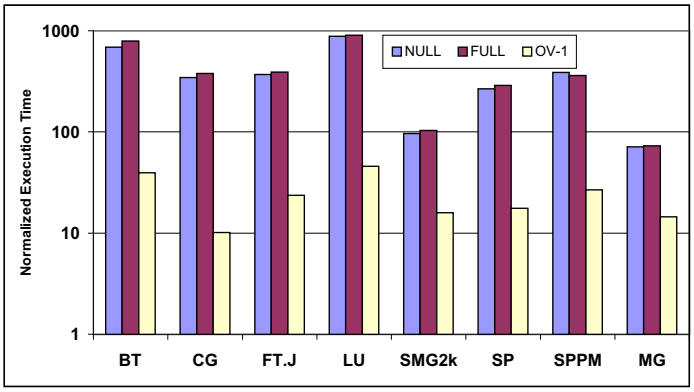


Figure 4.15: Execution Overhead Comparison

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.

4.11 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 [35].

Several frameworks simulate hardware and architecture state at the instruction level, which incurs considerable simulation overhead [43, 88]. 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 ([80, 7, 28]. MemSpy [69] and CProf [55] 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 [56]. SIGMA uses post-link binary instrumentation and online trace compression [30]. 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 4.15 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 [101] based on the SIMT multiprocessor simulation tool [100]. They use binary rewriting to extract the complete memory access trace using Valgrind [77] similar to the full-tracing approach we compare against in this work. 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 information to be correlated to the program source [70]. 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 [104]. 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 [12]. Buck *et al.* also compare different hardware mechanism for detecting uniprocessor memory hierarchy bottlenecks [11]. Satoh *et al.* study data-flow techniques to analyze data sharing patterns at compile time for OpenMP programs [92]. 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 [103]. We, in contrast, promote hardware-assisted sampling due to overheads resulting from software instrumentation in general.

4.12 Conclusion

In this chapter we presented 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.

Chapter 5

Hardware Profile-guided Automatic Page Placement for ccNUMA Systems

5.1 Summary

Cache coherent non-uniform memory architectures (ccNUMA) constitute an important class of high-performance computing platforms. Contemporary ccNUMA systems, such as the SGI Altix, have a large number of nodes, where each node consists of a small number of processors and a fixed amount of physical memory. All processors in the system access the same global virtual address space but the physical memory is distributed across nodes, and coherence is maintained using hardware mechanisms. Accesses to local physical memory (on the same node as the requesting processor) results in lower latencies than accesses to remote memory (on a different node). Since many scientific programs are memory-bound, an intelligent page-placement policy that allocates pages closer to the requesting processor can significantly reduce number of cycles required to access memory. We show that such a policy can lead to significant savings in wall-clock execution time.

In this paper, we introduce a novel hardware-assisted page placement scheme based on automated profiling. The placement scheme allocates pages near processors that most frequently access that page. The scheme leverages performance monitoring capabilities of contemporary microprocessors to efficiently extract an approximate trace of memory accesses. This information is used to decide *page affinity*, *i.e.*, the node to which the page is bound. Our method operates entirely in user space, is widely automated, and handles not only static but also dynamic memory allocation.

We evaluate our framework with a set of multi-threaded benchmarks from the NAS and SPEC OpenMP suites. We investigate the use of two different hardware profile sources with respect to the cost (*e.g.,* time to trace, number of records in profile) *vs.* the accuracy of the profile and the corresponding savings in wall-clock execution time. We show that long-latency loads provide a better indicator for page placement than TLB misses.

Our experiments show that our method can efficiently improve page placement, leading to an average wall-clock execution time saving of more than 20% for our benchmarks, with a one-time profiling overhead of 2.7% over the overall original program wallclock time. To the best of our knowledge, this is the first evaluation on a real machine of a completely user mode interrupt-driven profile-guided page placement scheme that requires no special compiler, operating system or network interconnect support.

5.2 Introduction

Cache-coherent non-uniform memory architectures (ccNUMA) constitute an important subset of current high performance computing platforms. Contemporary ccNUMA platforms, such as the SGI Altix, consist of a large number nodes, where each node has a small number of processors and a fixed amount of physical memory. All processors can access the same global virtual address space, but the physical memory is distributed across the entire system and coherence is maintained using hardware mechanisms.

In a ccNUMA system, accesses to virtual memory mapped on the same node as the requesting processor typically experience much shorter latencies than accesses to physical memory on a different node. We constructed an OpenMP micro-benchmark to evaluate access latency on our target platform, the SGI Altix. The program counts the processor cycles required to access physical memory on the local and remote nodes. The results are

shown in Table 5.1. We see that, on average, it takes more than twice as long to load from remote memory than from memory on the local node.

Table 5.1: Access latencies on the SGI Altix

Access Type	Average Latency (Cycles)	Standard Deviation
Local Node Memory	207	121
Remote Node Memory	430	176

In this paper, we focus on multi-threaded OpenMP benchmarks. Many of these programs are memory bound *i.e.*, the overall wallclock execution time of the program is significantly affected by the performance of the memory hierarchy. If the physical page placement is sub-optimal, *i.e.*, the bulk of the accesses are to pages whose physical memory has been allocated on a remote node, the program will take much longer to execute. On the other hand, an intelligent page-placement scheme, that allocates physical memory on nodes closer to the processors with most frequent accesses to a page, can reduce the average access latency leading to potentially significant wallclock time savings.

To effect this intelligent page placement, we must efficiently determine the overall memory access pattern of the program. In practice, even for reasonably-sized programs, it is difficult for programmers to know the best page placement for each page. Furthermore, on systems using "first-touch" page allocation, compulsory initialization of data elements (*e.g.*, from a file) in one thread can cause the page to be allocated permanently on a particular node. We have encountered OpenMP programs that have not been specifically tuned for ccNUMA environments and often initialize all data elements in the master OpenMP thread. This causes the bulk of the data space to be allocated in physical memory on only a single node, thereby drastically increasing the number of memory load instructions that access remote memory. Finally, even programs that specifically initialize ("touch") data in parallel on multiple threads can still achieve sub-optimal page allocation. This commonly occurs when the number of accesses to a particular page during the stable execution phase (*e.g.*, a single timestep) of the program may indicate a better page placement than the one effected by the parallel initialization with multiple threads.

To tackle these problems, we need an efficient whole-program analysis tool that considers the overall run-time memory access pattern of the program during its stable

execution phase and uses this information to decide the best page placement. In this paper, we contribute precisely such a scheme.

Our scheme works as follows. First, we execute a truncated one-timestep version of the program. We use the performance monitoring capabilities in existing microprocessors to efficiently extract an approximate trace of the memory accesses from all the active processors during this partial (truncated) run. We then use this access information to decide the best page placement, *i.e.,* the physical node on which a particular virtual page should be allocated ("affinity hints"). Finally, we run the complete program and use the affinity hints to allocate pages on the assigned physical node. The allocation is achieved by "touching" the target page from a processor on the assigned node, *i.e.*, by leveraging the default "first-touch" page allocation policy of the operating system. Our method handles both statically defined and dynamically allocated regions of memory. For statically defined memory regions (in the `bss` segment), the page touch takes effect at startup. For dynamically allocated regions, we delay the page touch till the region has been allocated.

Overall, we show that long-latency loads provide a better indicator for page placement than TLB misses and result in average wall-clock execution time savings of greater than 20% over all benchmarks, with a average one-time profiling overhead of 2.7% over the wallclock time for the complete original program. These results may make automatic page placement a cheap commodity without requiring user intervention.

The paper is structured as follows. First, we describe our page placement mechanism in detail. Then, we evaluate the placement mechanism with respect to the profile collection cost, the quality of the collected profile and the performance impact on the target program execution. We explore the use of two different profile sources, namely TLB misses and long latency loads, and the impact of different sampling intervals. Finally, we contrast our approach to related work and summarize our contributions.

5.3 Profile-guided Page Placement

Figure 5.1 shows our scheme. There are 3 distinct phases — *profile generation*, *affinity decision* and *profile-guided page placement*. In the profile-generation phase, we run a truncated version of the multi-threaded program (*e.g.*, a single timestep) and collect information about the memory access pattern for each thread. For the experiments

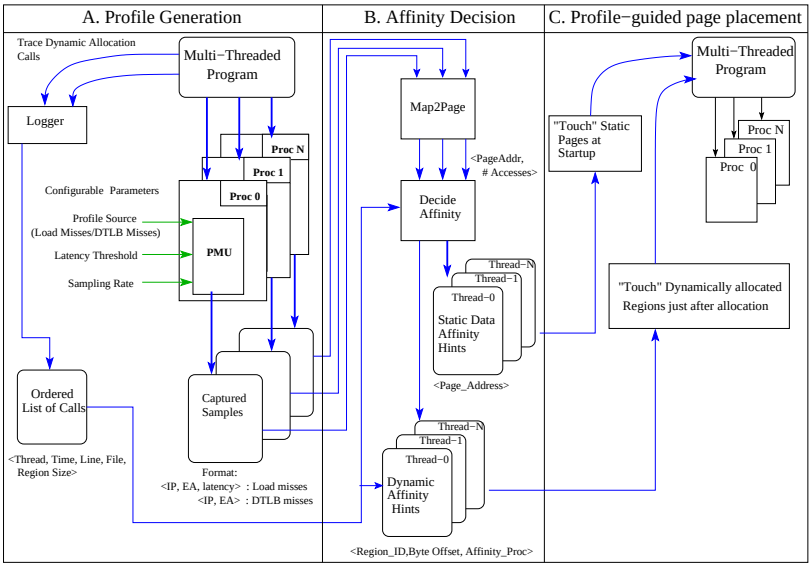


Figure 5.1: Automatic Profile-guided Page Placement

in this paper, we explicitly bind each OpenMP thread to a different processor using the `sched_setaffinity` primitive. We also intercept and log any dynamic allocation requests generated by the program. The collected information is used during the *affinity decision* phase to choose the most favorable mapping of pages to nodes, *i.e.*, the page affinity. Finally, we re-run the application and use the affinity information to force allocation of pages on their assigned affinity nodes.

We have automated our approach extensively such that user interaction is only required in three steps. First, a special header file transparently *wraps* allocation functions like malloc with calls to handler functions. Second, a call to an initialization function is placed at the very start of the program. This function effects page placement for statically-defined memory regions during profile-guided runs and initializes the hardware performance monitor during the profile collection run. Third, the user must identify the *stable execution phase* of the program and mark the phase with calls to handler functions. For example, in time-stepped programs, the stable execution phase is a single timestep. The idea is to collect a snapshot of the program's memory access patterns during a snippet of its stable execution phase and use that to guide page placement decisions. Next, our framework is described in more detail.

5.4 Profile Generation

We want to capture 2 types of profile information — memory accesses of each thread and calls to dynamic memory allocation.

Capturing Memory Accesses: We leverage the capabilities of the Itanium-2 performance monitoring unit (PMU) to capture an approximate trace of the memory accesses. We use the `libpfm` library to access the hardware counters of the processor [41]. The PMU operation is described in detail elsewhere [46]. In this paper, we use the PMU to capture two different types of memory access data — long latency loads and data translation lookaside buffer (DTLB) misses. A simplified view of the PMU operation for capturing long latency loads is shown in Figure 5.2. In this mode, the PMU supports selective tracking of load instructions based on a latency threshold. However, the PMU does not capture all

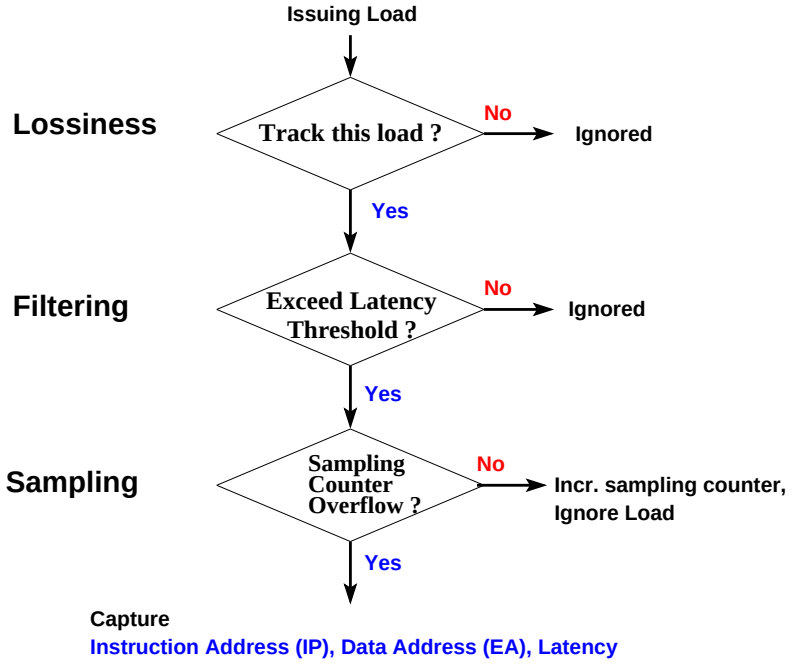


Figure 5.2: Simplified PMU Operation

long-latency loads that exceed the latency threshold. There are two reasons for this. First, due to hardware restrictions, the PMU can only track one load at a time out of potentially many outstanding loads. Second, in order to prevent the same data cache load miss from always being captured in a regular sequence of overlapped cache misses, the PMU uses

randomization to decide whether or not to track an issuing load instruction. Due to these reasons, the load miss trace that can be captured is *lossy*.

If a PMU-tracked load exceeds a user-configured latency threshold value, it qualifies for capture, otherwise it is ignored (*Filtering*). Since the access latencies increase monotonically for cache levels further away from the processor, the latency threshold allows selective capturing of the load miss stream (L1-D misses, L2 data load miss stream, etc.). Due to hardware limitations, the latency threshold can only be set in in powers of 2, with a lower bound of 4 cycles (*i.e.*, valid threshold values are 4 cycles, 8 cycles, 16 cycles, etc.) .

Each filtered load increments the PMU overflow counter. By appropriately initializing this counter, the user can vary the *sampling rate* for the captured long-latency load stream (*Sampling*). The Itanium-2 has special support to capture the exact instruction address (IP) and the corresponding data address being loaded (EA) for the sampled long-latency load. In contrast, counter-overflow based sampling on other processor architectures can give misleading instruction addresses for the missing load due to superscalar issue, deep pipelining and out-of-order execution [46].

The mechanism for capturing DTLB misses is similar, though there is no support for the latency threshold. Various specific sub-types of the DTLB misses can be captured (described in more detail in [46]). In this paper, we enable all types of DTLB misses for capture by selecting the corresponding `libpfm` event `DATA_EAR_TLB_ALL`. Each captured sample contains the address of the memory access instruction that caused the TLB miss and the accessed data address. This profile source includes DTLB misses caused by both loads and stores while the the long-latency capture mechanism described earlier only monitors load instructions.

Capturing Dynamic Allocation Information: For profile-guided page placement, we leverage the "first-touch" allocation policy that is used by SGI's Linux version for the Altix to allocate physical memory pages. To "touch" a particular virtual page address, we need to know the earliest point in the program at which such an address becomes valid. This requires the logging of memory allocation calls for each OpenMP thread.

The logger intercepts the program calls to `malloc`, `calloc` and `free`. We also log executions to Fortran `allocate` statements. The Itanium architecture has a high-resolution timer called the "interval timer counter" (itc). By logging the `itc` timestamp for each call and knowing the skew between the itc registers of each processor, a post-processing tool builds a unified ordering of allocation calls across all the threads.

5.5 Affinity Decision

Next, the approximate memory access trace and the dynamic memory allocation information is used to determine page affinity, *i.e.,* to decide the node on which physical memory should be allocated for a particular virtual memory page. The affinity decision module currently uses a simple decision criterion for mapping the page — the page should be allocated on the node that had the maximum number of accesses to that page. The idea is that by allocating pages closer to the most active requesting processors, the average latency of access can be reduced.

First, the accesses are grouped by page address, and the total accesses from all threads to each page are calculated by `map2page`, as shown in Figure 5.1. On our target platform (SGI Altix), each node has two processors that have identical latencies when accessing local physical memory. So the affinity decision module groups the accesses by each processor to calculate the per-node access count for each page. The page is recommended for allocation on the node with the maximum number of accesses to that page. The affinity decisions are generated differently for statically defined and dynamically allocated regions of memory.

Statically defined memory (*i.e.*, the `bss` segment) contains space for uninitialized global variables. The starting address and extent of the static region is determined at link time. The affinity decision module simply generates a per-node list of page address offsets that have affinity to that node. The first logical processor in a node is responsible for using these page offsets to issue the actual "first-touch" page placements during the final profile-guided program run.

Dynamically allocated regions pose an additional challenge since the starting address of the allocated region can and does change over multiple runs of the same program. For the benchmarks evaluated, two distinct dynamic memory allocation patterns were observed. Many programs had a small number of calls, each of which allocated a large chunk of contiguous memory. For such cases, we adjust the affinity page offsets relative to the starting address of the region. During the profile-guided run, just after the region is allocated, the affinity offsets will be used to "touch" the pages on the appropriate nodes.

Other programs had a large number of calls clustered in time, each allocating a very small region of memory (*e.g.*, NAS-2.3 MG). For such cases, we rely on the fact that these regions will most often be allocated contiguously in space. Since the memory access

trace is *lossy*, we observe in practice that we do not have even a single access record for many small allocated regions ("silent regions"). But we still handle these regions correctly because physical memory is allocated on *page* granularity, and we have trace records for other small regions whose page tends to include the silent region.

5.6 Profile-guided Page Placement

In this final phase, we re-run the original program and use the affinity information we generated in the earlier phase to guide our page placement decisions. At the time of writing, the operating system (SGI's Linux version for the Altix) does not support dynamic page migration at all. Instead, we leverage the existing "allocate-on-first-touch" policy to effect our page placement. This policy allocates physical memory for the virtual page on the node that first accesses ("touches") a data element on that specific node. Thus, to force page placement on a particular node, we "touch" the page by executing a load followed by a store to an address in the particular page from a processor on that node.

For this mechanism to succeed, we must touch the page before any other processor accesses that page. For static regions of memory, each processor reads its static affinity file on program startup and touches all the page address offsets listed in that file, as shown in Figure 5.1. All processors synchronize at a barrier after the touching phase, to ensure that no processor accesses a statically-defined page before the affinity hint for the page has been applied. Since the static allocation is done only once, at startup, it has minimal execution overhead.

The process for dynamically allocated regions is similar, except that we must delay the page touch till the target memory region is allocated. In this case, we know that, in a legal program, no other processor can access the allocated region before the allocation function (*e.g. malloc*) has completed. We take advantage of this to ensure that our first-touch scheme will effect the page allocation we want before any other processor touches the memory region. The idea is to insert a *wrapper* around the allocation call. The behavior of the wrapper is controlled by an environment variable. During unoptimized runs of the program, the wrapper does no work. During the profiling phase, the wrapper records the allocation request parameters (size of region, starting address, thread id, timestamp). The affinity generation phase tags each allocation request with affinity hints. Finally, during the profile-guided runs, the wrapper uses affinity hints to effect page allocation as follows.

When the wrapper is invoked, it first calls the real allocation function. The dynamic affinity hints provide information about which parts of this dynamically allocated regions should be allocated on which target processors. This information consists of a list of processor identifiers and the address offsets that need to be touched on these processors. For each such processor, the current thread reschedules itself on the target processor by using the `sched_setaffinity` function call. When the `sched_setaffinity` call returns, the thread is now executing on the target processor. Then the touch mechanism "touches" the given list of address offsets, thereby causing page allocation on the physical memory in the target processor's node. After all affinity hints for all processors for the dynamic region are processed, the thread re-schedules itself on its original processor.

This scheme is completely transparent to the user-level program, except for the addition of the wrapper function. However, it has high execution overhead. For every allocation request for which there are affinity hints for n processors, there are $n+1$ context switches (one switch to every target processor, and the final switch back to the original processor). We found during our evaluation (Section 5.7) that this scheme has substantial execution overhead, which erases the gain due to reduction in remote accesses for several benchmarks. The overhead can be reduced by a less transparent scheme that involves more effort on part of the user. A simple way to reduce overhead would be to *group* the touching effort for multiple dynamically allocated regions. For each group, there would be only one context switch for each processor in the list of affinity hints. The user would also need to insert additional synchronization to ensure that no thread begins accessing the dynamically allocated region before the touching has occurred (to prevent inadvertent page allocation). We leave this idea for future work.

There is one additional issue for dynamically allocated regions. In the benchmarks we evaluate, programs allocate memory only at the start of the program and do not free it till the end of execution. If, on the other hand, programs repeatedly allocate and free memory in the stable execution phase, then the effectiveness of the "first-touch" scheme would be reduced. This occurs because portions of the virtual address space may be "recycled" by the allocation function after they were initially freed, but the *physical* memory will only be allocated once on the node where the page of virtual memory was first touched. This is a limitation of using the first-touch mechanism. This issue could be solved if our operating system (Linux) supported dynamic page *migration*, which is not available at the present

time.[1] With migration support, the virtual pages in the dynamically allocated region could be simply *migrated* to the target processor given by the affinity hint.

5.7 Evaluation Framework

We described the scheme for profile-guided page placement in the previous section. In the following we present a cost versus benefit analysis as we vary the configurable parameters shown in Figure 5.1.

There are two configurable parameters that we shall vary — the choice of the *profile source* and the *sampling interval* for capturing memory access samples. The hardware provides two profile sources — long-latency loads and DTLB misses. The sampling interval provides a method to trade-off sampling overhead *vs.* the amount of profile data collected. For each profile source, we experiment with different sampling interval values (Sections 5.8 and 5.9).

As we vary these two parameters, the amount and type of profile information that we collect will change. How good are the affinity hints generated using these profiles? How effective are the affinity hints in reducing the overall wallclock execution time? To answer these questions, we need to compare the performance of these profiles with respect to the performance of affinity hints based on a *reference* profile. We call this reference profile the "maximum information profile". The reference profile answers this question: What affinity hints will be generated, if we knew as much as possible about the memory access pattern of the program? How much improvement in performance can be achieved using these hints? By comparing our profiles against the results achieved with the maximum information profile, we can clearly evaluate the tradeoff between profile collection cost and the optimization benefit.

Initially, we experimented with a software memory tracing tool to capture all memory loads and then used this access trace as the maximum information profile. However, this method had too much execution overhead for the benchmarks we evaluated. Instead, we configure the PMU with the lowest latency threshold setting (4 cycles) and the highest sampling frequency (1) and use the collected trace as the maximum information profile.

[1] Draft APIs for manual page migration have been proposed, and are expected to be available with future Linux systems.

Since the L1 cache hit access latency is 1 cycle, this corresponds to capturing a fraction of all accesses that miss in the L1 data cache.

In the discussion below, the "reference results" refers to the affinity hints generated using the maximum information profile. Similarly, the "target profile results" denote the affinity hints generated by the profile being evaluated.

Our evaluation has three aspects - the *profiling cost*, the *quality* of the collected profile, and the resulting *execution benefits*. The profiling cost is the cost of collecting the access trace, which is determined by the *size of the profile* and by the *execution overhead* inflicted on the benchmark during the profile collection phase. As the sampling interval is increased, both the profile size and the profiling overhead are expected to decrease.

For evaluating the quality of the profile, we shall compare the target profile results to the reference results using three different metrics. **Coverage** denotes the fraction of the pages in the *reference results* for which we have an affinity hint in the target profile results. The affinity node values for the page do not need to be the same between the reference and target profile results. **Accuracy** denotes the fraction of the pages in the *target profile results* that have the same affinity hint node value as the reference results. If the coverage value is low, it indicates that for large number of pages we simply do not have enough information in the target profile to generate an affinity hint. If the coverage value is high, we are confident that the target profile contains affinity hints for almost the same number of pages as the reference profile (though the affinity node *values* might be different).

In contrast, *accuracy* measures the stand-alone usefulness of the target profile. It answers the question: If the target profile were to be used to generate affinity hints, what fraction of the affinity hints are identical to those present in the reference results? If the accuracy value is high, it indicates that the target profile is as useful as the reference profile (though at a potentially much reduced overhead). On the other hand, a low accuracy value indicates that the target profile is potentially misleading in the sense that the affinity hints do not match the hints in the reference results.

Finally, the **Useful Fraction** metric combines the information from these two metrics. It measures the fraction of the affinity hints in the target profile that are not only present in the reference trace, but also have the *same* affinity node value.

The coverage, accuracy and useful fraction are computed as follows. Let

Ref = # hints in reference results;

Targ = # hints in target profile results;

C = # hints in target profile results that are also present in the reference results (though the affinity node values might not match);

A = # hints in target profile results that are also present in the reference results AND the affinity node values match. Then,

$Coverage = \frac{C}{Ref} * 100\%$

$Accuracy = \frac{A}{Targ} * 100\%$

$UsefulFraction = \frac{A}{Ref} * 100\%$

These three metrics each provide a different understanding of the profile characteristics. For example, a high accuracy value might still not indicate an *effective* profile if the coverage is low. This is because we simply will not have affinity hints for many pages (low coverage), but the few hints that we do generate are accurate (high accuracy). Similarly, a low useful fraction value could be either due to low coverage or to low accuracy of the hints. Thus, there is the need for all three metrics.

So far, we have seen the metrics for *cost* and *profile quality*. For assessing *profile benefit*, we measure two things - the net reduction in *remote accesses* and the reduction in *wallclock execution time*. The net reduction is compared by taking the difference between the metric values (remote accesses, wallclock execution time) between the original unmodified program run and the program run with our profile-guided page placement scheme.

The evaluation process works as follows. First, the target program is run for one time step and profile data is collected. This profile data is used to compute the affinity hints and the entire program is re-run using this profile data. Thus, the *profile cost* is the cost to capture the samples over one timestep of the program. On our platform, there is no easy way to measure the number of remote memory accesses generated by the program. Instead, we present an *approximate* measure of the reduction in remote memory accesses as follows. We set the PMU latency threshold to 512 cycles[2] and count the number of accesses that exceeded this threshold for the original program. The high latency threshold ensures that almost all loads that hit in cache or in local memory will be filtered out (though some remote loads may also be filtered out, as indicated by the latencies in Table 5.1). Then, we run the program with our page placement scheme and count the number of accesses exceeding the latency threshold (512) as before. The difference between the two values provides an

[2]Due to PMU limitations, the latency threshold can only be set in powers of 2. The next lower threshold (256 cycles) would not filter out a significant fraction of local memory loads, as indicated by the latencies in Table 5.1.

approximate measure of the net reduction in remote memory accesses. In practice, we have found this value to be quite consistent across multiple runs.

When comparing wallclock execution time, we compare the wallclock time for the complete run of the original program to the wallclock time of the program with profile-guided page placement, including the overhead of the page touching mechanism. During our experiments, we noticed that the execution time of the program varied measurably across runs. This may be due to several reasons. First, the difference in scheduler allocation of processors for the batch runs affects the degree of benefit obtained with profile-guided page placement (the benefit will be less if the allocated processors are closer). Second, all operating system calls on the Altix must go through a small collection of CPUs in the interactive login partition. Thus, the load on the interactive nodes affects the performance of the jobs running on the batch nodes. This is especially significant for the dynamic page touching mechanism, which potentially involves multiple context switches for a single affinity hint.

In order to account for this variability in execution time, we ran each benchmark for 6 times (5 times for BT). Each time, the profile-guided runs and the non-profile guided runs were executed on the same scheduler-assigned processor allocation. The wallclock execution time graphs show the average benefit obtained with each sampling interval. The error bars denote the confidence interval range for a 95% confidence interval.

Benchmarks: We use a set of 9 OpenMP benchmarks. This includes 7 out of the 8 NAS-2.3 benchmarks (excluding EP). The NAS benchmarks are C versions of the original NAS-2.3 serial benchmarks [4], provided by the Omni Compiler group [2]. We do not evaluate EP since it does not have significant sharing of data [67]. In addition, we also evaluate the 320.equake and 332.ammp benchmarks from the SPEC OMPM2001 benchmark set. These benchmarks have significant dynamic memory allocation, thereby putting our dynamic touching mechanism to the test.

All programs were compiled at the -O2 optimization level. All NAS benchmarks use Class C data sets, while the SPEC benchmarks use the reference data set. All experiments were carried out on a non-interactive (batch) allocation of eight processors. On our current platform, there are two processors per node. A total of four nodes were used. All programs were run with eight OpenMP threads. Each thread is bound to a separate processor using the `sched_setaffinity` primitive. OpenMP thread scheduling was set to static. Our hardware platform has Itanium-2 processors running at 1.5GHz, each with a 6

MB L3 cache, 256 KB L2 cache and 16 KB L1D cache.

For each program, we inserted markers delineating the start and end of the timestep. For 332.ammp, we disabled the pre-existing round-robin allocation of the "atom" element for the profile-related runs. However, we still compare the benefit metrics (wallclock time, number of remote accesses) against the original program. For the IS benchmark, we perform a one-time dynamic allocation for the `prv_buff1` array since the program failed to execute with the default stack allocation for this variable.

Out of the 9 benchmarks, 4 benchmarks — MG, 332.ammp, 320.equake, IS — utilize dynamic memory allocation. The remaining benchmarks operate with statically declared global arrays.

5.8 Evaluation with Long-latency Load Profiling

We evaluated the performance of our page-placement scheme with long-latency loads as the profile source. Figure 5.3 shows the performance using the cost / quality / benefit approach that was described in the last section.

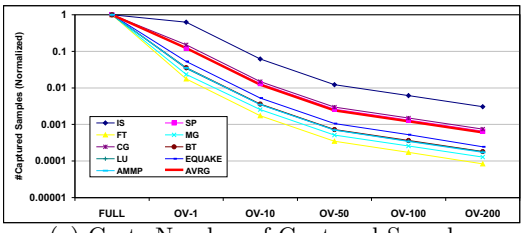
For these experiments, we fix the latency threshold in the PMU to 128 cycles. This filters out most of the load accesses that hit in the L1D, L2 and L3 caches. We select sampling intervals of 1, 10, 50, 100, 200 (OV-1 to OV-200 in the graphs).

Profiling Cost: The graph for cost comparison shows the cost for the "maximum information profile" (denoted as `FULL` in the graphs) and the results for each of the reduced sampling intervals. The reduced sampling results are normalized to the FULL profile values.

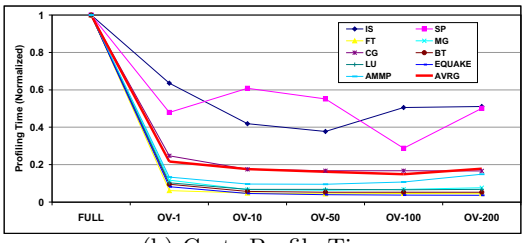
Number of Captured Samples: The number of accesses captured at OV-1, depicted in Figure 5.3(a), is about an order of magnitude lower than the FULL profile for most benchmarks (except IS). By keeping the latency threshold much higher (128 cycles instead of 4 cycles for the FULL profile) we filter out most of the loads that hit in cache. These loads can be ignored since they do not propagate past the cache to memory. Hence, they will not be affected by page placement.

With increasing sampling intervals, the total number of samples captured decreases linearly. At OV-200, the average number of accesses in the trace has been reduced by 1000 times over the FULL trace.

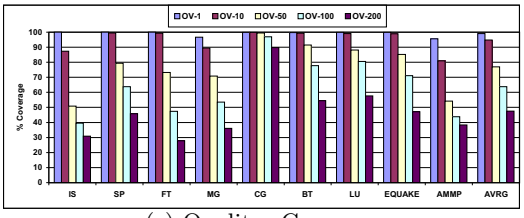
Profiling Execution Overhead: The absolute execution overhead for profiling



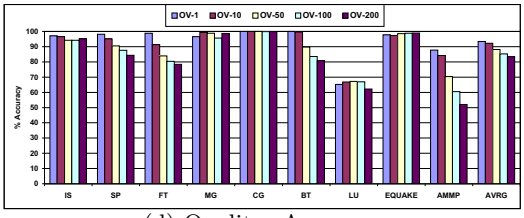
(a) Cost: Number of Captured Samples



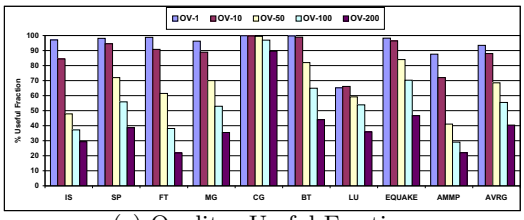
(b) Cost: Profile Time



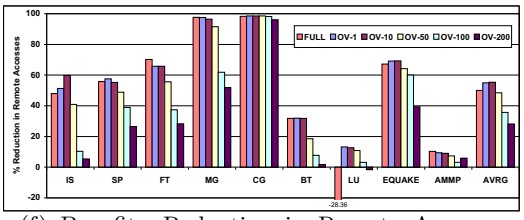
(c) Quality: Coverage



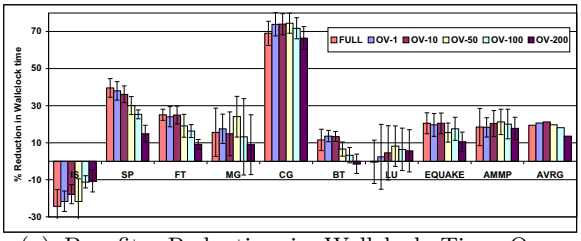
(d) Quality: Accuracy



(e) Quality: Useful Fraction



(f) Benefit: Reduction in Remote Accesses Over Original Program



(g) Benefit: Reduction in Wallclock Time Over Original Program

Figure 5.3: Evaluation with Latency threshold=128, Profile Source=Long Latency Loads

is extremely low, since it is sufficient to only a single timestep for the benchmarks that we considered, *i.e.*, the partial execution saves significant overhead over an execution of the entire benchmark without any loss in accuracy for the benchmarks studied.

On average, over all benchmarks, we measured the execution overhead for profiling a single timestep at OV-1 to be 2.7% of the overall original program execution time.

The *relative* profiling execution overhead (compared to `FULL`) is shown in Figure

5.3(b). We see that the overhead *flattens* out with increasing sampling intervals. This indicates that the profile collection cost does not dominate the time to execute the timestep. The results show that OV-1 or OV-10 are the "sweet spot" values for the sampling interval, since increasing sampling intervals beyond that point does not reduce overhead by much.

On average, profiling execution overhead at OV-1 is about 20% of the FULL profile cost, with the exceptions of `SP` and `IS` that have lower savings.

Profile Quality: As the sampling intervals increase, the size of the profile collected will tend to decrease. This has an effect on the quality of the profile, *i.e.*, the *coverage*, *accuracy* and *useful fraction* metrics. The maximum values of all these metrics is 100%.

Coverage: Figure 5.3(c) depicts the coverage results for different sampling intervals. At OV-1, the average coverage is 99% indicating that we have affinity hints for almost all of the FULL profile pages. The OV-10 coverage still remains high at 94%. After that, we observe a noticeable decline in coverage at sampling intervals of OV-50 and beyond. The average coverage falls from 94% at OV-10 to 76% at OV-50 and finally to 47% at OV-200. Thus, at OV-50 and higher, we simply do not have enough profile data to generate affinity hints for page placement for a significant number of pages.

Accuracy: The accuracy values, depicted in Figure 5.3(d), are very close across sampling intervals for each benchmark. Also, accuracy remains uniformly high across increasing sampling intervals for all benchmarks (except for LU). This is very encouraging as it indicates that even with a reduced number of accesses, the affinity node recommendations match the recommendations given by the FULL profile for most of the affinity hints generated. LU's behavior is explored in more detail later.

Useful Fraction The useful fraction is the fraction of the FULL profile affinity hints that are present and have the same affinity node value in the target profile results. A high useful fraction indicates that we are obtaining almost the same results as the FULL profile results, with much smaller profile input data.

The average useful fraction, depicted Figure 5.3(e),is high for OV-1 (93%) and OV-10 (87%). From OV-50 to OV-200, the metric degrades from 68% to 40% on average. This trend occurs because the *coverage* values fall with increasing sampling intervals while the accuracy remains steady. The degradation is much more pronounced for benchmarks like `IS`, `FT` and `MG` whereas there is almost no degradation for `CG`.

Profile Benefits: We explore the impact of the page placement scheme on two metrics: (1) the number of remote accesses generated by the program and (2) the wallclock

execution time of the program.

Reduction in Remote Accesses: Figure 5.3(f) shows the net reduction in the number of remote accesses for the full-program run using automatic profile-guided page placement *vs.* the original program. The figure compares the reduction in remote accesses using the FULL profile, *vs.* the reduction achieved at latency threshold 128, and the different sampling intervals.

For all but one case (LU:FULL profile), there is a net reduction in the number of remote accesses. Almost all the remote accesses for CG and MG are eliminated as shown by a 98% and 97% reduction at OV-1 for CG and MG, respectively.

Other benchmarks also have significant reduction in remote accesses. The average reduction at OV-1 is 60% and decreases significantly from OV-50 (48%) to OV-200 (28%). OV-10 appears to be the sweet spot. The average reduction is, in fact, slightly higher for OV-10 (55%) than OV-1 (54%). Only LU shows a 28% *increase* in remote accesses when using the full profile. This anomaly of LU is discussed in more detail later.

Reduction in Wallclock Execution Time: This is the most important measure to assess the overall benefit as it indicates the performance improvement of an application with profile-guided placement compared to the original unmodified program. Figure 5.3(g) shows the improvement in wallclock time. As described before, the error bars represent the 95% confidence interval range. The ranges for MG, LU and IS are large, indicating that these programs have more variable execution times.

Except for IS, every other benchmark shows a reduction in wallclock execution time. The average reduction is 21% at OV-1. CG achieves exceptionally large savings with over 73% shorter executions at OV-1. Many other benchmarks (SP, FT, MG, Equake) also achieve greater than 15% reductions.

With increasing sampling interval, the wallclock improvements tend to decrease, though the magnitude of decrease is program-dependent. CG does not show much degradation with increasing sampling intervals, but there is a noticeable degradation with SP between OV-10 and OV-200.

IS represents an exceptional case where the wallclock execution time *increases* with profile-guided page placement. We determined that the cause of the degradation is the cost of the page-touching mechanism for dynamically allocated regions. Each hint on a dynamically allocated region potentially represents at least two context switches — one to switch to the target processor and "touch" the page and the other to switch back to

the processor that originally requested the allocation. (Note that we bind each OpenMP thread to a different processor. Hence, we refer to processors instead of threads here.) With increasing sampling intervals, fewer dynamic hints are generated (as coverage falls). This reduces the overall overhead on the target. Thus, we see less degradation in wallclock execution time for IS with increasing sampling intervals.

Similar to IS, the potential wallclock savings for other programs with dynamic memory allocation (MG, Equake, AMMP) are also affected by the overhead of the touching mechanism. Given that over 98% of the remote accesses for MG are eliminated by page placement, the wallclock reductions for MG would increase even further with a more optimized touch mechanism.

The LU Anomaly: LU represents an anomalous case. For this benchmark, the affinity hints generated by the full profile do not match the affinity hints generated by the other profiles (OV-1 to OV-200). This causes low accuracy and useful fraction values, as seen in Figures 5.3(d) and 5.3(e). Furthermore, using the full profile leads to an *increase* in the number of remote accesses (Figure 5.3(f)) while OV-50 leads to a 10% decrease in remote accesses. The corresponding wallclock time reduction is *higher* for OV-50 (8% improvement) than that of the full-profile results (0% improvement).

The underlying cause is as follows. The affinity node values differ between the full profile and the OV-1 profile (and higher sampling interval profiles) for parts of the large `rsd` global static array. The full profile uses the lowest possible latency of 4 cycles to sample the address trace. This captures all possible loads, irrespective of whether the loads hit in cache or not. For the pages of the `rsd` array that have different affinity hints in the full and OV-1 profiles, most of the accesses on the affinity node given in the full-profile are hits in the local caches. Hence, the affinity decision is different from the OV-1 profile-based decision (which filters out the cache hits). First, we observe that loads which are hits in cache will not be affected by page placement decisions. Second, the full-profile based page placement, in fact, *worsens* the average access latency for cache misses since the corresponding pages are allocated on a node that only has infrequent cache misses for those pages. This explains the *increase* in the average number of remote accesses for the full-profile results compared to the OV-1 based experiment. Thus, the average wallclock time improvement is lower for full-profile than for OV-50 in this case.

Conclusions: Long-Latency Profiling: 1) Overall, we observe that the size of the profile data at OV-1 is one-tenth the size of the FULL profile on average. With increasing

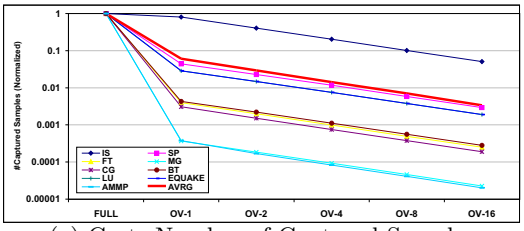
sampling intervals (OV-1 to OV-200), the profile size decreases linearly. 2) For most benchmarks, the execution overhead of profile collection decreases sharply from FULL to OV-1, yet it does not decrease significantly with larger sampling intervals (OV-10 to OV-200). Thus OV-1 or OV-10 appears to be the *sweet spot* for profile collection. 3) With increasing sampling intervals, the coverage drops significantly, which indicates insufficient profile information to generate affinity decisions for many pages. 4) Nevertheless, the *accuracy* of the profile information does not degrade significantly with increasing sampling intervals. 5) A significant reduction in the wallclock execution time and the number of remote accesses is possible with profile-guided page placement. However, for programs with dynamic allocation, the page touching mechanism is expensive and adversely affects wallclock execution time. A more optimized touching scheme should lead to even better wallclock reductions for these programs. 6) For one benchmark (LU), using the reference profile (`FULL`) actually resulted in a *degradation* of performance. For this benchmark, the *filtering effect* of the high latency threshold used by the target profiles (128 cycles) removed loads that hit in the cache and resulted in a more accurate picture of which pages are frequently accessed by which processors. Thus, using the full memory access trace may actually result in sub-optimal page placement in rare cases. For all other benchmarks, the reference profile almost always had the maximum (or close to maximum) performance benefits, *i.e.*, reduction in remote accesses and wallclock time.

5.9 Evaluation with Data TLB Misses Profiling

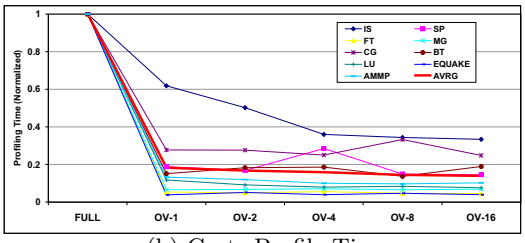
Figure 5.4 depicts the results using data TLB misses as the profile source obtained with PMU support. We evaluate results for sampling intervals values of 1, 2, 4, 8 and 16 (denoted OV-1 to OV-16 in the graphs). For the discussion below, we shall refer to the results presented in the last section using long-latency loads as the profile source as the *load-based results*. In the following, we describe the DTLB miss results and contrast them with the load-based results.

Profile Cost: As before, the cost metrics are compared against the cost incurred for the "maximum information profile" (denoted as `FULL` in the graphs).

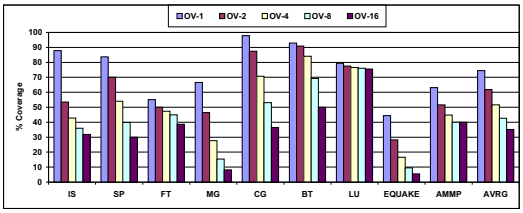
Number of Captured Samples: The average number of samples captured at OV-1 is less than one-tenth of the number of samples in the full profile, as seen in Fig-



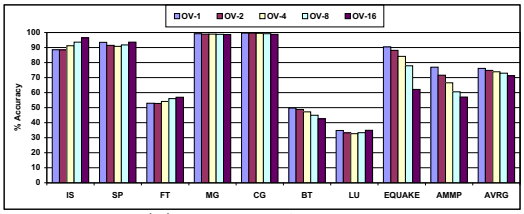
(a) Cost: Number of Captured Samples



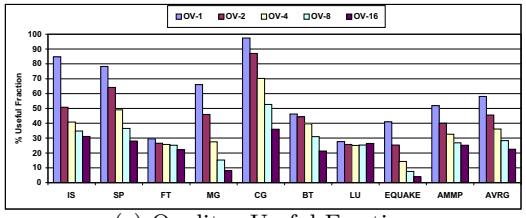
(b) Cost: Profile Time



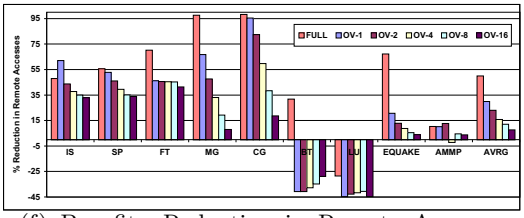
(c) Quality: Coverage



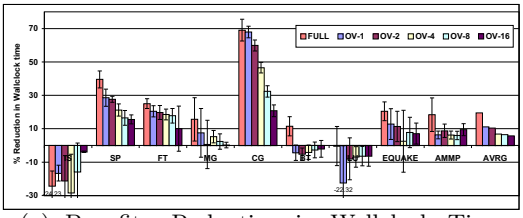
(d) Quality: Accuracy



(e) Quality: Useful Fraction



(f) Benefit: Reduction in Remote Accesses Over Original Program



(g) Benefit: Reduction in Wallclock Time Over Original Program

Figure 5.4: Evaluation with Profile Source=Data TLB Misses

ure 5.4(a). With increasing sampling intervals (OV-1 to OV-16), the number of captured samples decreases almost linearly.

In contrast to the load-based results, the difference between FULL and OV-1 tends to be program-dependent. Ammp and MG have more than 1000 times less profile data at OV-1 compared to FULL while IS has almost the same number of samples as FULL.

Profiling Execution Overhead: The results for the relative profile overhead, depicted in Figure 5.4(b), are similar to load-based results. The average execution overhead for trace collection at OV-1 is 18% of the FULL profile's cost. With increasing sampling intervals (OV-1 to OV-16), the execution overhead is not significantly reduced.

Profile Quality: As before, we evaluate the three quality metrics of *coverage*, *accuracy* and *useful fraction* shown in Figures 5.4(c), 5.4(d) and5.4(e), respectively.

Coverage: The average coverage at OV-1 (74%) is sharply lower than the average coverage at OV-1 in the load-based results (99%), as depicted in Figure 5.4(c). This is due to significantly lower coverage values for FT, MG, LU, Equake and Ammp, as compared to the load-based results. With increasing sampling intervals, the coverage begins to degrade significantly, except for LU. Coverage falls from 74% at OV-1 to 35% at OV-16.

The low coverage values indicate that we have insufficient information to generate page affinity hints for a significant number of pages. The problem is more acute for the DTLB case than for the load-based results, as indicated by the lower coverage values. Low coverage lessens the effectiveness of the page-placement scheme resulting in a reduced potential for performance benefits.

Accuracy: The results in Figure 5.4(d) indicate that accuracy is benchmark-dependent. For most benchmarks (except Equake and Ammp), the accuracy values for increasing sampling intervals are similar. This indicates that accuracy is less sensitive to reduction in the size of the profile trace.

We also observe sharply lower accuracy for FT, BT, LU and AMMP compared to the load-based results. This indicates that page-affinity decisions based on DTLB misses do not agree with affinity decisions based on the FULL trace or long-latency load-based results.

Useful Fraction: Due to the sharply lower coverage (FT, MG, LU, Equake, Ammp) and lower accuracy (FT, BT, LU), the useful fraction values are also significantly lower than for the load-based results. The average value at OV-1 is 58% compared to 93% at OV-1 with long-latency loads as the profile source.

With increasing sampling intervals, the useful fraction value tends to fall significantly for most benchmarks. The average useful fraction degrades from 58% at OV-1 to 22% at OV-16.

Profile Benefits: We have seen that the coverage, accuracy and useful fraction for DTLB-based results are significantly lower than their load-based counterparts for most

benchmarks. This will impact the performance benefits obtainable with profile-guided page placement. Figures 5.4(f) and 5.4(g) show the reductions in remote accesses and overall wallclock execution time, respectively.

Reduction in Remote Accesses: As before, the reduction in remote accesses using profiles obtained at different sampling intervals is compared to the reduction obtained with results based on the full profile (marked FULL) seen in Figure 5.4(f).

Two benchmarks, BT and LU, experience an *increase* in remote accesses with DTLB-guided page placement. The increases are significant (more than 30%) and occur with all sampling intervals. In comparison to the load-based results, the reduction in remote accesses is much lower for many benchmarks, especially for MG (98% *vs.* 67%) and Equake (69% *vs.* 20%). The average reduction of remote accesses is 29% at OV-1, which is much lower than the 54% average reduction at OV-1 for the load-based results.

Reduction in Wallclock Execution Time: As with remote accesses, the DTLB miss-based scheme generally performs worse than the long-latency load-based mechanism. The average wallclock reduction at OV-1 is 11% for DTLB misses (see Figure 5.4(g)) *vs.* 20.6% for the load-based results.

IS, LU and BT show an increase in execution time with DTLB-guided feedback. CG has the maximum improvement (67%), while improvements reduce sharply for MG (17% *vs.* 7%), Ammp (18% *vs.* 6%) compared to load-based results at OV-1.

Conclusions for DTLB-Profiling: 1) Overall, the cost of profile collection is similar for both DTLB misses and long-latency load-based schemes. 2) The coverage and accuracy for DTLB-based results are significantly lower for DTLB-based results compared to the long-latency load-based results. 3) Due to sharply lower coverage and accuracy, the useful fraction values are also low. This indicates that DTLB-based affinity decisions are not representative of decisions that would be made with the full profile. 4) The performance benefits (reduction in remote accesses and wallclock time) are also much lower for DTLB-based results. 5) The profile costs for both DTLB misses and long-latency loads are similar, but the quality of the profile and the resulting performance benefits are much larger for long-latency load-based profiles compared to the DTLB miss based profiles.

We conclude that DTLB misses are not a good candidate to decide page placement. This shows that, for the benchmarks we considered, DTLB misses do not correlate well with the relative volume of loads from a processor to a particular memory page. This could occur, for example, if the program has few DTLB misses but a large number of cache misses going

to memory. Then, the information about the *frequency* of accesses to each page is lost if we only consider the DTLB misses (since repeated accesses to the same page will tend to hit in the DTLB). [3]

5.10 Related Work

Tikir and Hollingsworth describe a dynamic user-level page migration scheme based on an approximate trace of memory accesses obtained by sampling the network interconnect [104]. This is the closest related work. The trace is used for deciding page affinity. Pages are dynamically migrated using the `madvise` system call. In contrast, we focus on profile-guided page placement leveraging the simpler "first-touch" page allocation policy of the operating system. [4]

Our method uses a different profile source (long-latency loads or DTLB misses) with varying sampling intervals. Our method is simpler in that it is *processor-centric*. More specifically, we not require special network instrumentation support, we only rely on the ability of the PMU to *time* load accesses. Because their approach is *network-centric*, *i.e.*, the hardware counters are embedded in the network interconnect and do not distinguish between different processes, only one application can use them at a time. In contrast, there is no such restriction with our approach. In addition, our mechanism is interrupt-driven, *i.e.*, the PMU raises an interrupt only when the sampling counter overflows and generates virtual addresses directly. In contrast, their method must *poll* the network interconnect counters to collect a trace of physical addresses, which must subsequently be mapped to virtual addresses using a separate system call.

Finally, our page hints are *abstracted*, *i.e.*, they are relative to the starting address of the region (static or dynamic). Touching is deferred till the region is actually allocated. Thus, the affinity hints are potentially *portable across platforms* in that hints generated on one platform can be used on another if it supports first-touch page placement. We intend to explore this potential in future work.

[3] Another possible scenario is a large number of DTLB misses with few cache misses going to memory. In this case also, the DTLB trace will not be representative of the relative distribution of load requests to a page from each processor.

[4] In the future, our approach can be extended in a straightforward way to eliminate the need for a separate profiling run by *migrating* pages. This depends on proposed future extensions in Linux to support dynamic page migration under user control.

Nikolopoulos *et al.* describe a user-level dynamic page migration scheme that uses per-page hardware reference counters that capture the frequency of accesses from each node to a particular page [81, 83]. The method depends on the compiler for identifying the pages of virtual memory using whole program analysis. In contrast to our method, they do not handle dynamic memory allocation. In addition, we don't require any compiler or operating system support, and our page-placement mechanism is completely transparent to the target program (*i.e.*, no explicit calls are necessary for page placement).

Verghese *et al.* describe a simulation-based kernel-level implementation of dynamic page migration [106]. They consider both number the of load-misses to a page and the number of data TLB misses as profile sources. In our work, we found data TLB misses to be less effective for deciding the best page placement, which confirms results presented in their work.

Other approaches to kernel-level dynamic page migration and replication are discussed in Noordergraaf *et al.* [84] and in Bolosky *et al.* [6]. In contrast, we operate completely in user-space and leverage the simpler first-touch page allocation policy to steer page placement at region initialization.

Bull and Johnson study the tradeoffs between page migration, replication and data distribution for OpenMP applications on the Sun WildFire system [13]. In their study, they find that page replication performs better than page migration and static data distribution.

Lastly, the hardware mechanism for capturing long-latency loads and DTLB misses is described in the Itanium-2 manual [46]. In previous work, we used this facility in conjunction with software rewriting to efficiently obtain a lossy load/store trace and exploit its information to analyze the coherence behavior of OpenMP programs [67].

5.11 Conclusion

In this work, we developed and evaluated a low-cost whole-program analysis tool that considers the overall run-time memory access pattern of the program during its stable execution phase. It uses this information to decide the best page placement. The novelty of our work also lies in the exploration of hardware-assisted performance monitoring techniques, completely in user mode, without any special compiler, operating system or network interconnect support.

Our technique operates as follows. First, we execute a truncated one-timestep version of the program. We leverage performance monitoring capabilities in existing microprocessors to efficiently extract an approximate trace of the memory accesses from all the active processors during this partial (truncated) run. We then use this access information to decide the best page placement, *i.e.,* the physical node on which a particular virtual page should be allocated ("affinity hints"). Finally, we run the complete program and use the affinity hints to allocate pages on the assigned physical node. The allocation is achieved by "touching" the target page from a processor on the assigned node, *i.e.*, by leveraging the default "first-touch" page allocation policy of the operating system. Our method handles both statically defined and dynamically allocated regions of memory. For statically defined memory regions (*i.e.*, the `bss` segment), the page touch is effected at startup. For dynamically allocated regions of memory, we delay the page touch till the region has been allocated.

Our framework is currently constrained to work with the "first-touch" page placement policy, as dynamic page migration is not supported on Linux at the present time. Due to this, we cannot do effective page allocation for programs whose memory access patterns *change over time*, *e.g.*, adaptive mesh refinement (AMR) codes, and programs with multiple execution phases. Also, the first-touch based scheme would lose effectiveness on programs which frequently allocate and free memory during the stable execution phase (none of the programs in this study show this behavior). When page migration support is added to Linux, we shall overcome both these limitations.

Overall, we show that long-latency loads provide a better indicator for page placement than TLB misses that results in average wall-clock execution time savings of more than 20% over all benchmarks. with an average one-time profiling cost of 2.7% over the overall original program wallclock time. The low overhead may make automatic automatic page placement a cheap commodity without requiring user intervention.

Chapter 6

PFetch: Software Prefetching Exploiting Temporal Predictability of Memory Access Streams

6.1 Summary

CPU speeds have increased faster than the rate of improvement in memory access latencies in the recent past. As a result, with programs that suffer excessive cache misses, the CPU will increasingly be stalled waiting for the memory system to provide the requested memory line. Prefetching is a *latency hiding* technique that tackles this problem. If the address of the memory line that misses in cache can be predicted sufficiently in advance, it can be prefetched into the cache before it is accessed, reducing the effective latency of that access.

In this work we propose a novel software-only data prefetching scheme that works at the instruction level and exploits predictability in the access stream to prefetch memory lines accessed in the future. Working at the instruction level gives us a *global* view of memory accesses patterns across function, module and library boundaries. Conceptually, our scheme

monitors the memory locations being by loads and stores, as well as their *contents*. It tries to find instances of *predictability* such that the address of a load miss can be pre-determined from a limited number of past accesses.

We make the following contributions in this work. First, we present a novel prefetching strategy that *unifies* and generalizes a number of past approaches that each target a specific source of address predictability. Specifically, our scheme unifies all these past approaches: next-line prefetching, self-stride prefetching, "intra-iteration" stride prefetching and same-object prefetching. In addition, it extends and generalizes the SPAID scheme for pointer and call-intensive programs. Second, we present a new threshold-based approach that addresses the issues of *prefetch accuracy*, *prefetch timeliness* and *prefetch redundancy*. Third, we evaluate our scheme both with a cache simulator and on a real machine where we evaluate it with hardware performance counters.

Overall, we demonstrate that a significant reduction in L1 cache misses can be achieved for several benchmarks on a real machine with our approach. However, the resulting reduction in processor cycles is lower than anticipated, and we detail several possible causes explaining this phenomenon.

6.2 Introduction

In the recent past, processor speeds have been increasing at a much faster pace than improvements in memory access latencies. As a result, the cost of a cache miss in terms of processor cycles keeps increasing. Due to limited out-of-order window sizes of contemporary processors, a load miss in the second or futher levels of cache will typically stall the processor as it runs out of parallel instructions to process. Consequently, the overall wallclock time for applications is often dominated by the efficiency of their memory access patterns.

Prefetching is a *latency hiding* strategy that attacks this problem. If the address of a load that misses in cache can be predicted sufficiently early, then the corresponding memory line can be prefetched into the cache well in advance of its access. If the prefetch completes before the load, then the erstwhile miss would be transformed into a hit.

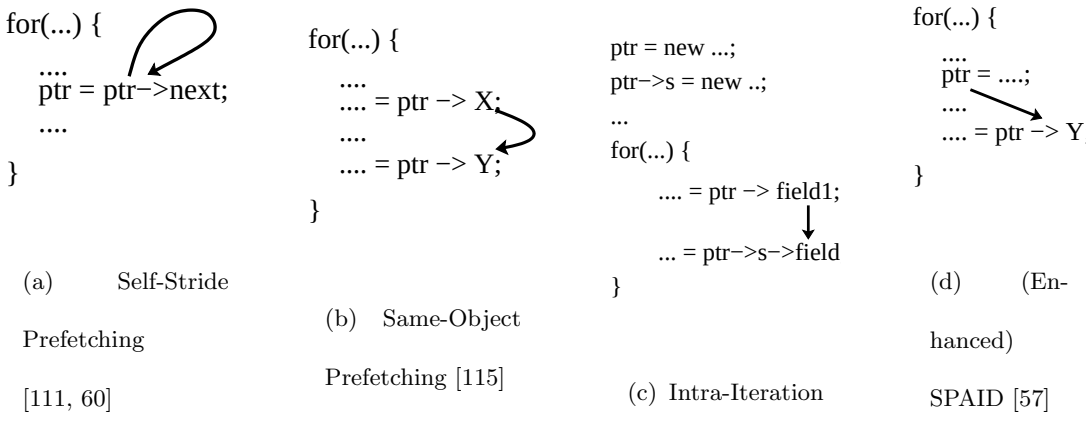
In this work we focus on integer intensive programs that typically contain few regular array accesses. Software prefetching algorithms for arrays accesses are well estab-

lished(*e.g.*, [73]), and we do not target these in our approach[1].

Conceptually, our scheme monitors the memory locations being accessed by loads and stores, as well as their *contents*. It tries to find instances of *predictability* such that the address of a load miss can be pre-determined from a limited number of past accesses. Our scheme is based on offline analysis using profile feedback. First, the program is run with a small training data set and an annotated trace of memory accesses is extracted. This trace is analyzed offline for detecting predictability and a set of *prefetch predictors* is generated. The prefetch predictors are used to place explicit software prefetch instructions directly in the assembly code of the program. In contrast to earlier hardware solutions ([76, 117]), our scheme operates completely in software and we present results from an implementation on a contemporary processor platform.

Our scheme has several advantages over past work. By examining the overall memory access streams of the executing program, we get a *global* view of a program's memory access pattern across function and module boundaries. This is hard for a static compiler to achieve, especially with irregular integer programs due to aliasing and input-dependent control flow. At the same time, our analysis is powerful and general enough to encompass and unify multiple separate approaches from past work. Specifically, our scheme unifies all these past approaches: next-line prefetching [96, 50], self-stride prefetching [111, 60, 86, 33], "intra-iteration" stride prefetching [45] and same-object prefetching [114]. In addition, it extends and generalizes the SPAID scheme [57] for pointer and call-intensive programs. A detailed discussion of related work is presented in Section 6.5. Examples of the access patterns targeted by our scheme are shown in Figure 6.1, along with the past work that addresses that pattern. The bold arrows depict the source and target of prefetching. Our scheme does not target each pattern *specifically*, it turns out that all these patterns can be effectively handled by a standard approach to analyzing memory accesses. Figure6.1(a) shows an example of self-striding that typically arises in pointer chasing code where consecutive instances of the data structure tend to be placed at regular offsets from each other. Figure 6.1(b) shows "same-object" prefetching, where different fields of an object are frequently accessed close together in time. Since field layout is statically determined, the address of any field can be computed if the address of any other field of the same object is known. Figure 6.1(c) demonstrates "intra-iteration" stride prefetching. In many cases,

[1]Our baseline executable for performance comparison has compiler-inserted software prefetch instructions for array accesses.



(a) Self-Stride Prefetching [111, 60]

(b) Same-Object Prefetching [115]

(c) Intra-Iteration Stride Prefetching [45]

(d) (Enhanced) SPAID [57]

Figure 6.1: Example Patterns

data structures are allocated at the same time as their children. As a result, the address of the children fields can be predicted from the parent object's address or the address of some other child. Inagaki *et. al* demonstrate that this occurs frequently with Java programs [45]. Finally, Figure 6.1(d) shows a generalized scenario targeted by a SPAID-like scheme. If the field dereference (ptr-¿y) is sufficiently distant from the pointer initialization, then the field can be prefetched in advance. Lipasti *et. al* only target pointers being passed as function arguments at call sites and also do not consider *offsets* from the pointer, as typically occurs with the dereferencing of a *field* of an object or a structure. We have generalized SPAID to consider *any* pointer load or store as a potential prefetch source. In addition, we consider load misses to a contiguous region of memory *around* the pointer as prefetch targets. This supports the common case of field dereferences that the original SPAID scheme did not consider.

We make the following contributions in this work:

- We present a novel software-only prefetch scheme that finds miss address predictability by monitoring the memory accesses of the program at the instruction level.
- Our approach unifies and encompasses several separate past efforts that each targeted different sources of predictability.
- We enhance and generalize the SPAID scheme that targets pointer dereference misses

at function call sites.

- We also present novel measures to filter prefetch predictors based on *prefetch accuracy*, *prefetch timeliness* and *prefetch redundancy*. We are not aware of any past work that uses this approach.

- We implemented our scheme on a real machine and evaluate its performance with hardware performance counters.

6.3 Framework

We are looking to exploit the *predictability* of memory access streams for reducing the number of load cache misses. We monitor the address (EA) generated by load and store instructions. In addition we also monitor the *contents* of the memory location being accessed (EA_Contents) for loads/stores that might be accessing pointers.[2] Conceptually, for each load miss, we consult the recent history of memory accesses, constrained by a window size, to see if the load miss address was *predictable* using the captured EA, EA_Contents information of a retired dynamic instruction P. If so, then the load miss can be potentially transformed into a hit by inserted prefetch instructions that use the EA/EA_Contents value of P to prefetch data into the cache. In addition, we also try to address *prefetch timeliness* by checking whether the load miss is too *close* temporally to the predictor instruction P. If the load miss is found to be too close, the prefetch would not be useful.

For a target benchmark we proceed through the following stages. First, we generate the assembly source code for the benchmark and use an annotation tool to instrument memory access instructions. In addition, we insert instrumentation to maintain a pseudo "instruction counter" that conceptually increments after every instruction is issued[3]. We call this variable the *instruction distance (Inst_Dist)* counter. The idea is to tag each traced memory access with the Inst_Dist counter value. Later, during analysis, this helps in improving prefetch timeliness by detecting whether prefetch predictors may be issued too close (temporally) with respect to the target load miss instruction, in which case the

[2]Our experiments were performed on the Power architecture, where pointers are usually accessed using 32-bit `lwz` and `stw` instructions and their variants.

[3]We reduce the overhead by appropriately updating this counter only at basic block boundaries and before memory access instructions.

prediction would not be useful. Our experiments were performed on the Power architecture. We considered annotating the memory accesses with the values from the hardware high precision timebase register in place of our software instruction distance counter. However, reading the timebase register is too expensive (1000 cycles for back-to-back reads of the timebase register), which significantly distorts the actual number of cycles between memory accesses. On other architectures (*e.g.*, Itanium2) the cost of reading the high precision timer is much lower. This may make it feasible to use the timebase register in place of the software instruction distance counter.

In the second step, we run the instrumented program and collect the memory access trace. Third, we analyze the trace using our framework and generate *prefetch predictors*. Each prefetch predictor is a tuple <IP, EA/EA_Contents, Delta>.'IP' is the unique identifier for a load or store instruction. 'EA/EA_Contents' describes whether the effective memory address accessed by the instruction or the contents of that memory address should be used for prediction. Finally, 'Delta' is a constant value that denotes the offset from EA/EA_Contents to the memory line that needs to be prefetched. In the final phase, we insert a prefetch snippet just after the target instruction indicated by the prefetch predictor IP. The prefetch snippet issues a prefetch for the address EA/EA_Contents + Delta using the "Data Cache Block Touch" (`dcbt`) instruction. All these steps are completely automated.

Even with train data sets, the number of loads and stores in the full trace is very large. We therefore implemented *bursty tracing* to reduce the number of samples in the trace. The bursty tracing used a duty cycle of 10% with each burst containing 2 million accesses [4].

We shall now describe our analysis in more detail. Figure 6.2 shows the steps in the analysis. We maintain a cache simulator "filter" that models the first-level L1D cache. The filter tags each memory access as a hit or a miss. Only load misses are targeted for prefetch prediction. All memory accesses are considered as potential prediction sources. For each memory access described by <IP, EA, EA_Contents, Inst_Dist>, we generate candidate predictors off both EA and EA_Contents. The idea is to keep track of a fixed contiguous region of memory *around* the EA and EA_Contents address, for a certain number of accesses in the future. If any of these future accesses are load misses in this region, then those load

[4] In other words, we captured 2 million accesses and then ignored the next 18 milllion accesses.

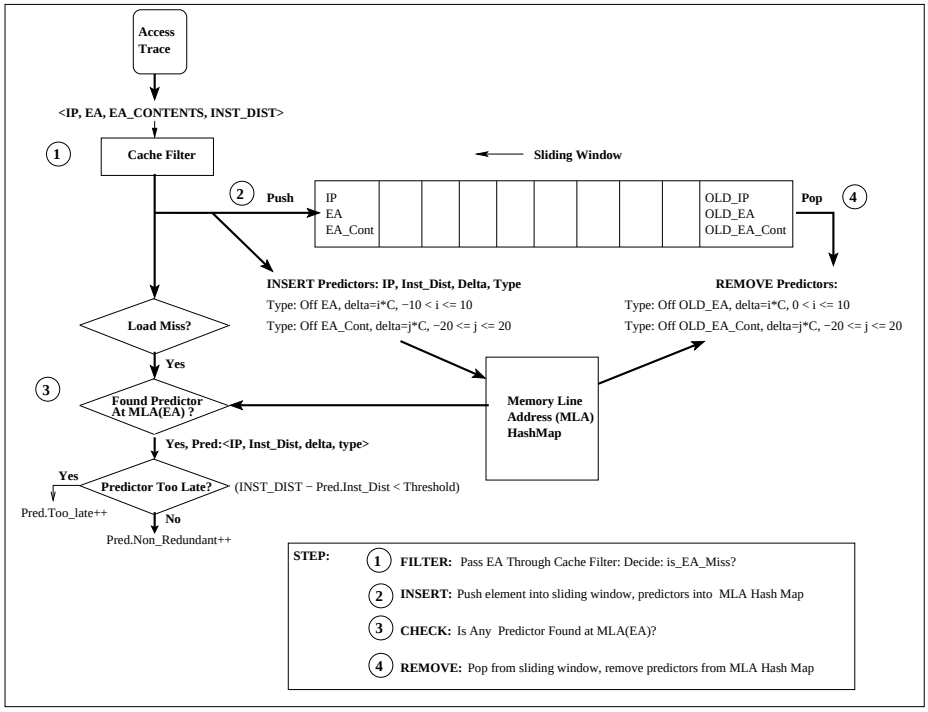


Figure 6.2: Overall Framework

misses can be prefetched using the value of EA or EA_Contents and a constant offset value *Delta*, as described before. The size of the contiguous region is configurable. We empirically determined that region sizes of 10 and 20 cache lines in each direction (positive/negative) gave good results for EA and EA_Contents, respectively[5]. A smaller region potentially reduces the number of load misses detected as suitable for prefetching, but with the benefit of reduced analysis overhead.

How long should we keep the predictors around? This is a configurable parameter and is implemented by a *sliding window* scheme. The length of the sliding window decides the number of future accesses that the predictors will be active. The trace record (EA, EA_Contents, IP) is put into the sliding window when the candidate predictions are generated. When the trace record pops out of the sliding window, the predictors are removed. The candidate predictions are hashed by their memory line addresses and stored in the *Memory Line Address Hash Map (MLA map)*. For each load miss, the load miss address is checked in the MLA map. If there are existing predictions for the load miss address, each

[5]In Figure 6.2 `C` is the cache line size.

predictor is considered further for validity. The Inst_Dist values of the predictor and the load miss are checked. If the difference in instruction distance is below a certain threshold, the prediction is classified as "too late". Otherwise the prediction is considered useful (Non_Redundant).

After the entire trace has been processed, we prune the predictors using various thresholds. First, we consider the *accuracy* of predictions. Predictors are considered highly accurate if the predicted cache line is either already in the cache or is accessed within the sliding window before the predictor was removed (when popping off accesses from the end of the sliding window). This is known as the *seen_ratio*.

seen_ratio = (# predicted line seen in sliding window or already in cache) / (# predictions)

The idea is to reduce the overhead of *useless prefetches* by only selecting predictors with high accuracy. Useless prefetches are very expensive because they bring in data that is seldom accessed and may evict frequently accessed memory lines from the cache, in addition to the overhead of executing the prefetch snippet.

Our second threshold addresses the issue of *prefetch timeliness*. The Inst_Dist difference between a predictor and its target load miss denotes the number of dynamic instructions issued between them. This is a conservative lower bound on the number of *processor cycles* between these two events, since it does not account for multi-cycle instructions such as loads that miss in cache. If this difference is lower than a certain threshold, we consider the predictor to be "too close" temporally to the target load instruction that missed. Hence, the prediction is not useful. The *too_late_ratio* is calculated as:

too_late_ratio = (# predictions classified as too late/ # predictions)

If the too_late_ratio is above a certain value, the predictor is pruned.

Finally, we attempt to reduce the number of *redundant* prefetches. Consider, for example, a structure S with three elements A, B and C that reside in different cache lines with the fields always accessed in the order S.A, S.B and S.C. If S.C is a miss, then the load miss address can be predicted using both S.A and S.B. However, the second predictor (off S.B's EA) is redundant and should be pruned. The redundant predictors are pruned as follows. First, the set of predictors pruned using the other thresholds discussed above is generated. Then, the trace is re-processed from the start and passed through a new cache filter. At the end, the redundancy_ratio for each predictor is calculated as follows:

redundancy_ratio = (# predicted line already in cache / # predictions)

If the redundancy_ratio is above a certain threshold then the predictor is pruned.

At the end of the analysis, we have obtained a set of predictors that are highly accurate, have good timeliness and are highly relevant. These predictors are inserted back into the assembly source code as described earlier. Our current scheme is based on annotating assembly code, though our framework can be implemented in the back-end of a compiler as well. The prefetch snippets as well as the instrumentation snippets need at least two free registers. In our current implementation, we reserve two registers using a compiler flag[6] when generating the target program's assembly source. This requirement can be removed by selecting dead registers using live variable analysis [60].

Our scheme is not targeting load misses incurred by regular array accesses. Prefetching algorithms for such accesses are well established. Our compiler (gcc) is able to generate prefetches for such array accesses. In our analysis we take into account the effect of this compiler-generated prefetching in the following way. When tracing memory accesses, we also trace the prefetch instructions inserted by the compiler. During analysis, the effect of these prefetch instructions is simulated by the cache filter. It is important to do this because, otherwise, our analysis might generate prefetch predictors for array accesses that are also targeted by the compiler-generated prefetch. Our predictors would be redundant in this case.

6.4 Experiments

We evaluated our framework for a set of memory intensive benchmarks shown in Figure 6.1. The benchmarks have been selected from the SPEC CPU 2000 [40], Ptrdist [3] and Olden [18] suites, except for boxedsim [20]. We selected benchmarks that had significant L1D cache miss rates. Smaller data sets were used for the training phase, while larger data sets were used for measuring execution benefits. For the boxedsim benchmark and the benchmarks from the Olden and PtrDist suites we increased the data set sizes from their default values to make the programs run sufficiently long.

[6] We used gcc for our experiments. The corresponding flags for gcc are -ffixed-r15 -ffixed-r16

Table 6.1: Benchmarks and data sets

Benchmark	Suite	Train Data Set Arguments	Ref Data Set Arguments
181.mcf	SPEC CPU2000	train/inp.in	ref/inp.in
300.twolf	SPEC CPU2000	train	ref
255.vortex	SPEC CPU2000	bendian.raw	bendian1.raw
175.vpr	SPEC CPU2000	net.in arch.in place.out dum.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2	net.in arch.in place.out dum.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2
197.parser	SPEC CPU2000	2.1.dict -batch < train.in	2.1.dict -batch < ref.in
boxedsim	-	-n 500 -t 0.75	-n 1000 -t 1.0
ft	PtrDist	10000 40000	10000 40000
bh	Olden	8192 1	65536 1
bisort	Olden	10000000 1	20000000 1
em3d	Olden	10000 100 75 1	1000000 100 75 1
health	Olden	5 3000 1	5 4000 1
mst	Olden	4096 1	8192 1
treeadd	Olden	24 1	26 1
tsp	Olden	2000000 1	3000000 1
voronoi	Olden	500000 1	10000000 1

6.4.1 Procedure

All experiments were conducted on a multi-user p655 Power4 SMP system. The system has 8 processors, but we used only one of them since our programs are single-threaded. Each Power4 processor has a 16 KB 2-way associative L1D cache, 1.5 MB 8-way associative unified L2 cache and a 32 MB 8-way associative L3 cache. The L1D and L2 cache lines are 128 bytes wide.

For each program the compiler was used to generate assembly code. We used the gcc compiler with high optimization settings (-O3 -mpower -fprefetch-loop-arrays[7]). In addition, as stated before, we also reserved two registers for our prefetch snippet in the generated assembly code using the flags "-ffixed-r15 -ffixed-r16". The executable built from this assembly source is our baseline for performance evaluation. In the profiling/analysis pass, this assembly source was instrumented for bursty tracing and the generated traces were analyzed using our framework. The generated predictors were inserted into the assembly source that was assembled and linked to create our modified executable for evaluation.

We only had access to a shared multi-user Power4 platform. The shared usage caused variability in our performance measurements. In order to address this problem

[7]The "-fprefetch-loop-arrays" directs the compiler to insert explicit prefetch (`dcbt`) instructions for loop-resident array accesses.

each measurement run was repeated for 10 times, and the *minimum* values were chosen for comparison.

6.4.2 Self-Striding Comparison

Past related work on data prefetching has focused on *self-striding* [60, 111]. We implemented a self-striding scheme and compared its performance to our framework. The implementation of the self-striding scheme is based on Wu *et al.'s* scheme [111]. For each load instruction, the load access trace generated is examined. Up to 4 most frequent strides are selected. Strides that account for less than 25% of the accesses at that instruction are pruned. We keep track of the average instruction distance B (Section 6.3) between consecutive accesses at an access point. If the access point is in a loop, this represents an approximate estimate of the number of cycles that the loop body takes to execute. If the cache miss penalty is L cycles, the prefetch multiplier K is calculated as:
K = min($\lceil\ L/B\ \rceil$, C) , where C is the maximum distance multiplier. We use L=120, C=10. 120 cycles is the latency for an L2 miss on the Power4. The value for C is chosen to be similar to past work reported in Wu *et al.* [111] to permit a fair comparison.

The prefetch predictor has a distance of K*stride and is inserted into the assembly source code as described before.

6.4.3 Measurement Metrics

We use two metrics — L1D load cache misses and the total number of processor cycles executed by the program. We used hardware performance counters to measure these two metrics with the `hpmcount` performance monitoring tool (events PM_LD_MISS_L1 and PM_CYC, respectively). The performance measurements were undertaken with a different and larger data set compared to the training data set used for tracing and analysis (Figure 6.1).

In addition, we also measured the L1D load cache misses on a cache simulator using only the train data set for both the original and predictor-inserted variants. The simulator is event-based (*i.e.*, it does not model timing), and the cache size is same as the as the real machine's L1D cache (32 KB). It also does not model the hardware stream prefetcher available in the Power4 platform. In spite of its limitations, the simulator output

provides a useful indication of the magnitude of *potential* miss rate savings possible, as will be shown.

6.4.4 Analysis

Table 6.2: Analysis Parameters

Parameter	Value
seen_threshold	0.85
too_late_threshold	0.6
redundancy_threshold	0.95
inst_dist_threshold	64
sliding_window_size	450
ea_locality_lines	10
ea_contents_locality_lines	20

Table 6.2 shows the values of the parameters used during our analysis. Thus, predictors were pruned if they did not satisfy these conditions: the predicted memory line was "seen" at least 85% of the time, the prediction was too late more than 60% of the time or the prediction was redundant (*i.e.*, target memory line was already in cache) more than 95% of the time. The sliding window size was 450 and candidate predictors were generated for up to 10 and 20 cache lines in each direction for EA and EA_Contents respectively.

Figure 6.3 shows the percentage reduction in L1D cache misses as reported by the simulator processing the traces from the train data set. It gives a useful indication of the benchmarks where our technique is applicable. Figure 6.4 and 6.5 show the percentage reduction in (a) L1D cache misses and (b) processor cycles, respectively, on the real machine as reported by the hardware performance counters with the reference data set.

Consider the simulation results shown in Figure 6.3. In 8 benchmarks (out of 14), our scheme is able to reduce the L1D cache miss rates by 5% or more, with more than 20% reduction in 5 benchmarks (mcf, parser, em3d, mst, treeadd, tsp). Self-striding performs significantly worse in 6 of these benchmarks (vortex, parser, boxedsim, em3d, mst, tsp), shows comparable performance in 2 (mcf, treeadd) and is significantly better for FT. FT is discussed in more detail below. For the other benchmarks, the difference in performance with respect to self-striding can be attributed to the fact that our scheme targets multiple different sources of predictability (Section 6.2) while self-striding only focuses on the

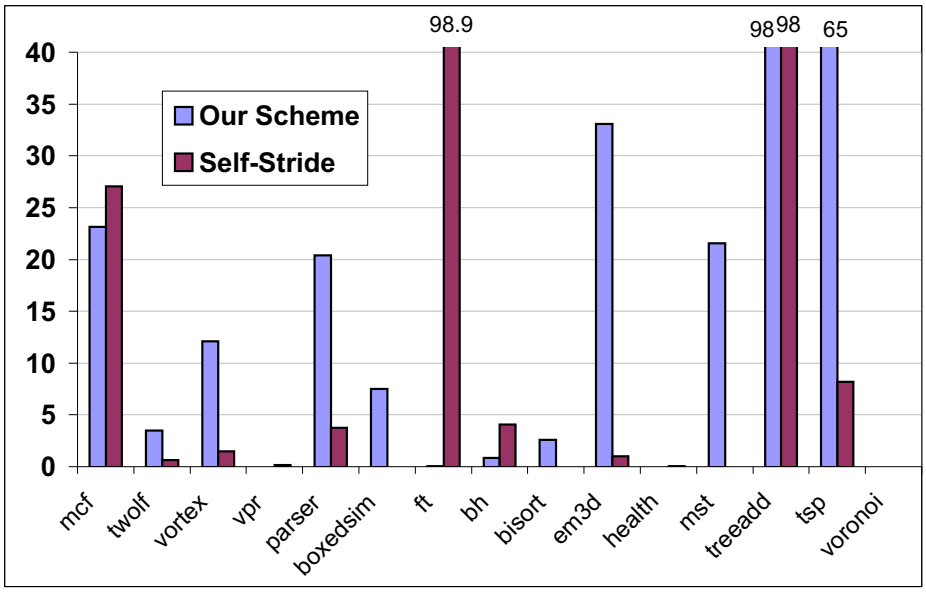


Figure 6.3: Simulator: % Reduction in L1D load misses (Training data set)

regularity of the access stream at individual access points.

Consider the actual reduction in L1D misses obtained with the reference data sets in Figure 6.4. We observe that there are significant savings for many benchmarks, but the values are uniformly lower as compared to the simulator predictions (except for vortex, where they are higher). For our scheme, vortex and tsp show significant reduction ($> 30\%$) while mcf, boxedsim, em3d and mst exhibit appreciable reduction ranging from 4.9% to 17%. In comparison, self-striding has significantly worse performance in 4 of these benchmarks (vortex,em3d, mst, tsp), comparable performance in 2 (mcf, boxedsim) and performs significantly better for FT. Some benchmarks that show significant improvement with the simulator do not show a corresponding improvement on the real machine with the larger ref data set (parser, mst, treeadd). The potential reasons for this discrepancy are discussed below.

Figure 6.5 shows the corresponding reduction in processor cycles. The performance results are mixed. 6 benchmarks (mcf, vortex, vpr, bisort, em3d, tsp) show appreciable reduction, ranging from 2% to 7.6%. However, 4 benchmarks (parser, boxedsim, ft, treeadd) experience slowdowns ranging from -2% to -12%. Except for FT, self-striding always performs worse for all benchmarks, with no benchmark showing an improvement of 2% or more. For self-striding, mcf achieves significant L1D miss reduction, but almost no processor cycle

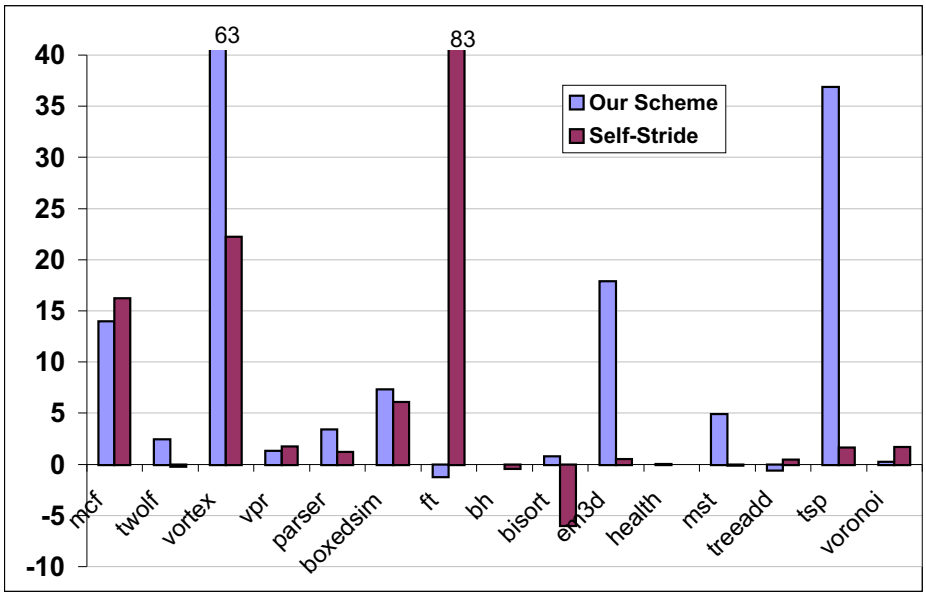


Figure 6.4: H/W: % Reduction in L1D load misses (Reference data set)

reductions. We suspect that the cause is suboptimal instruction scheduling for the prefetch snippet, as discussed below.

We have seen that for many benchmarks, the simulator results are much better than the results on the actual machine. In addition, for several benchmarks the L1D miss reductions on the real machine do not correspond to a reduction in processor cycles. In fact, some benchmarks show a degradation in performance. What could account for these anomalies? To attain a definitive answer, we would need a cycle accurate simulator that models our target (commercial) Power4 processor. There are several plausible explanations:

- **Simulator**: Our simulation results are expected to be *optimistic*, since the simulator is event-based and does not model prefetch timeliness. In addition, it does not currently model the hardware stream prefetcher present in the Power4 platform. Some of the predictable prefetches may also be recognized by the hardware prefetcher, reducing the observed L1D miss savings on the real machine.

- **Prefetch snippet Cost**: We do not account for the cost of the prefetch snippets. This cost can overwhelm the benefits of reduced misses if there are many redundant prefetches.

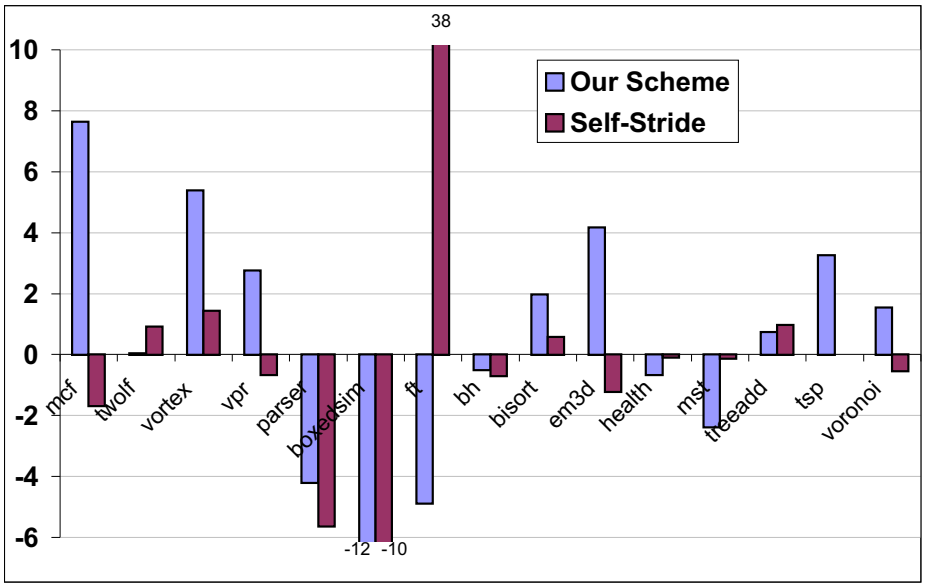


Figure 6.5: H/W: % Reduction in processor cycles (Reference data set)

- **Prefetch scheduling**: Our current approach is based on annotating the assembly source code. In the course of our experiments, we found that scheduling of the prefetch snippets has a big impact on the overall processor cycles. Implementing our scheme in the backend of the compiler would solve this problem.

6.4.5 FT

FT illustrates a potential weakness of our scheme. Self-striding performs exceptionally well for FT, while our scheme did not find significant prefetch predictors. In this benchmark, there is a single load access in a tight pointer-chasing loop that accounts for the bulk of the load misses. The access point exhibits a regular stride of -120. Self-striding is able to detect this stride and generates a prefetch predictor distance of -1200 (since the maximum distance multiplier, C, is 10) off the effective address (EA) of the access point. Recall that our sliding window analysis only generates predictors for a *limited* region around the access point's EA. In our experiments this was set to 10 cache lines, so the predictors generated had prefetch distances of -1152, -1024,...-128. However, all these prefetch predictors were *classified as "too late" because of the tight nature of the pointer-chasing loop nest and, hence, they were rejected.*

Thus, in this case, self-striding was able to generate prefetch predictors while the sliding window analysis failed to do so. The cause was that the high instruction distance threshold (64) was *too pessimistic* — because it did not consider the number of cycles the processor was stalled due to a cache miss, but only considered the actual number of instructions issued between the candidate predictor and the target dynamic load instruction.

In future experiments we shall evaluate the impact of lowering the instruction distance threshold from 64. This would definitely help for FT but may impact other benchmarks adversely.

6.5 Related Work

Sair *et al.* performed a simulation-based study to classify program load misses into *next-line*, *stride*, *same-object* and *pointer* misses [91]. A significant fraction of the load misses was found to be of one of the first three types for many of the SPEC benchmarks. It is precisely these misses that are targeted by our framework. Hardware implementations of next-line and stream prefetching have been proposed earlier [96, 50]. The Power4 platform has a stream-based hardware prefetcher. However, this prefetcher needs a "warm-up" period of a certain number of misses to consecutive cache lines before it starts to prefetch the stream. Hur et al propose a memory-side prefetcher for the Power5+ system that targets such short-length streams [44]. In contrast, our approach is completely software-based and uses explicit prefetch instructions to target such streams successfully.

Stride-based prefetching has been explored both in hardware and software. Unlike our scheme, much of the past work has focused on the striding regularity of *individual* access points, *i.e.*, they do not consider predictability among accesses from different load/store instructions. Wu *et al.* [111] and Luk *et al.* [60] describe such software self-striding schemes. Hardware stride prefetchers have also been proposed [86, 33] and implemented in the Pentium microprocessor. As shown in this work, our scheme performs significantly better than self-stride prefetching in isolation for most benchmarks. This is because our approach is able to detect multiple other sources of predictability in addition to most cases of self-striding. However, in a few situations, self-striding may perform better than our scheme (*e.g.*, for FT). Zhang and Torrellas propose a hardware scheme to group fields or objects that are used together, and prefetch all these at the same time [115]. In comparison, our scheme will

also recognize groups of field accesses that occur together and target them for prefetching if they miss frequently in cache. However, our scheme is implemented completely in software.

Lipasti *et al.* propose SPAID, a prefetching heuristic for pointer and call intensive programs [57]. SPAID inserts prefetches for the target of pointer parameters at call sites, on the assumption that the pointed object will be dereferenced soon. Our scheme extends SPAID in that *any* pointer load is analyzed as a potential prefetch candidate, not just the pointers at call sites. In addition, when looking for predictable cache misses we look at a large contiguous area of memory *around* the candidate pointer, while SPAID will only prefetch at most one cache line (the one pointed to by the pointer). Our experiments indicate that our approach is better because pointers pointing at objects or structures are often used to deference *fields* of an object that spans multiple cache lines. Finally, our detailed trace information allows much better pruning of harmful or useless prefetches. The dynamic instruction distance allows us to model *prefetch timeliness* for SPAID-like prefetch predictors, pruning away those that would be too late to be useful. The redundancy threshold prunes away predictors whose targets are most often already in cache.

Inagaki *et al.* present a software prefetching approach for targeting both "inter-iteration" (self-striding) and "intra-iteration" predictability (predictability across instruction points) [45]. This is the closest related work. During just-in-time compilation of a hot Java loop, the load dependence graph is constructed. An interpreter runs the first few iterations of the loop and attempts to determine stride patterns among the different loads in the dependence graph. This information is used to insert software prefetches, and speedups of up to 20% are reported on some benchmarks. Our scheme shares the goal of determining address predictability among multiple memory access points. However, our offline approach allows us to conduct a much more detailed analysis by accounting for prefetch timeliness and prefetch redundancy. Our scheme targets misses from *all* loads, not just the loop-resident ones. Finally, our scheme can potentially *group* multiple prefetches because it considers *memory lines* and not just the strides between two loads in a pair — since multiple fields typically share the same cache line.

Hardware schemes have been proposed that assess predictability in misses generated from different access points [76, 117]. In contrast, our approach is completely in software and is therefore more portable.

Finally, there are other prefetch schemes that are complementary to our approach. Software prefetching for regular array accesses in loop intensive programs is well estab-

lished [73]. We do not target these misses, but focus on other sources of address predictability in irregular integer-intensive programs. Prefetching of pointer chains, for example using Markov predictors, has been previously proposed in both hardware [49, 27, 26, 89] and software [22, 90]. Our scheme does not address this problem. Finally, helper-thread based approaches leverage additional hardware contexts to prefetch data for the main program [58, 51]. In contrast, we focus on prefetch instructions inserted *inline* in the target program that can be single-threaded or multi-threaded.

6.6 Conclusions and Future Work

In this chapter, we have presented a novel software-only prefetch scheme that exploits predictability in the memory access stream to create prefetch predictors. Our approach unifies and extends several past approaches that each targeted a different source of address predictability. Our approach is fully automated, portable and uses novel threshold-based mechanisms for addressing prefetch accuracy, prefetch timeliness and prefetch redundancy.

We have evaluated our scheme on a real machine as well as a cache simulator. We have shown that significant reductions in L1D load cache misses are achievable on real hardware with our approach. However, the performance improvements in terms of processor cycles has been lower than anticipated (and, in some cases, negative). Based on experience with microbenchmarks, we conjecture that *prefetch scheduling* has a significant impact on the overall processor cycles. In future work, we hope to port our framework to the back-end of a compiler, which will allow us to schedule prefetch snippets. In addition, we hope to port our framework to other memory constrained architectures such as the Itanium2 platform.

Chapter 7

Conclusion

In the past chapters, we have described several performance analysis and optimization frameworks. All these ideas are united by the common theme of using memory access traces as the primary input for analysis.

With these tools, we have demonstrated that that memory traces are a rich source of information about program behavior and can be used for analyzing and optimizing many different performance aspects of software programs, thus answering the primary thesis problem statement in Section 1.1. With respect to the other objectives listed in the thesis problem statement, we make the following observations.

1. **What are the different methods to obtain the program's memory access trace?** We explored three ways to obtain memory traces — in software, using *dynamic binary instrumentation*, in hardware, exploiting *performance monitoring unit (PMU) support*, and *hybrid* schemes that use a combination of software annotation and alternative hardware support (such as high precision timing registers).

2. **What are the *quality* versus *overhead* tradeoffs between these trace generation methods?** We compared the tradeoff between trace quality and generation overhead in several of our frameworks. In general, software tracing is the most expensive, but allows all trace accesses to be captured. Thus it provides the maximum information for trace-based analyses among all tracing methods. PMU-based tracing is usually much faster and allows filtering of loads by cycle thresholds directly in hard-

ware. However, the resulting trace is lossy and the PMU has restrictions on types of memory accesses that can be traced (*e.g.*, stores cannot be logged by the Itanium2 PMU). Finally, hybrid tracing allows software to control the degree of lossiness but still drastically reduces the volume of captured accesses by using high precision timer information.

3. **What are the different performance analysis and optimization frameworks that are feasible using memory traces?** We discussed five frameworks based on memory traces that cover a variety of performance analysis and optimization techniques.

4. **What are the *pre-requisites* in terms of trace information and detail that these analyses entail, and is it possible to satisfy them with memory traces and additional instrumentation?** Each framework placed different pre-requisites on the input trace. Some frameworks worked well with PMU-generated lossy traces (*e.g.*, automated page placement) while others experienced degradation in some specific areas (*e.g.*, false-sharing with PMU-generated traces). The prefetching framework, in contrast, *required* non-lossy traces because of its reliance on modelling processor caches.

   Most frameworks required auxiliary information in addition to memory traces. For the prefetching framework, we also added extra instrumentation to keep a count of the dynamic instructions issued and used it for prefetch timeliness. For ccSIM and METRIC, we extracted symbolic information from the executable and used it for abstracting the generated event metrics, *e.g.*, the instruction to source location mapping, the address and size of global variables, *etc.*. Finally, for ccSIM we also obtained OpenMP synchronization information by instrumenting the compiler-generated functions that correspond to OpenMP compiler directives.

Now we shall revisit each framework and highlight our contributions.

7.1 METRIC: Memory hierarchy analysis for single-threaded benchmarks

In this work, we demonstrated a trace-based framework for analyzing the memory performance of single-threaded programs. We used a novel instrumentation strategy based on dynamic binary rewriting that allow extraction of partial access traces from executing programs. We contributed a framework for selective instrumentation of load and store instructions on-the-fly. We also introduced a new light-weight compression strategy to efficiently compress the trace online. We demonstrated that our compression strategy achieves better compression than the previous known state-of-the-art algorithm for a majority of the benchmarks (7 out of 12), and has comparable compression for the rest.

Our tool generates a rich set of metrics tagged to source code constructs that provide detailed information about the program's memory behavior. These include per-reference cache metrics, evictor information and stream metrics generated by the compression algorithm. We demonstrated the use of these metrics to detect and understand memory access inefficiencies with several use-cases that are hard to achieve with static compiler analysis.

7.2 ccSIM: Source-Code Correlated Cache Coherence Characterization of OpenMP Benchmarks

In this work, we described a new, *user-level* approach to analyzing coherence bottlenecks. Our framework is the first tool to analyze the sharing pattern of multithreaded OpenMP programs. We introduced a new cache coherence simulator, ccSIM. ccSIM obtains coherence metrics and tags them to high level source code constructs using symbolic information embedded in the binary.

Our results indicated a good match between our simulations and the observed hardware performance counters for coherence events. The detailed metrics generated by ccSIM were useful in understanding the *flow* of data across processor caches. This enabled the detection, understanding and resolving of potential sharing bottlenecks.

We used ccSIM to obtain significant run-time savings in several large real-world

benchmarks. The optimizations based on ccSIM data ranged from coarsening of access granularity over data alignment to call parallelization, critical section removal with privatization and prefetching.

7.3 Hybrid Hardware/Software Coherence Analysis

In this work, we presented two novel hardware-assisted approaches to determine cache coherence bottlenecks. We compared these two methods to our earlier full-trace based method used in our ccSIM work.

Our first method leveraged the performance monitoring unit (PMU) to efficiently obtain filtered lossy memory access traces (PMU-based tracing). We described a new source annotation approach that transforms store instructions into equivalent instruction snippets that can be logged by the PMU. Our second method (targeted tracing) leveraged the high precision timer register to perform load filtering in software, which allows greater control over the degree of lossiness but at relatively larger overhead compared to the PMU-based method.

We evaluated both these methods for trace collection overhead and trace sizes and compared the accuracy of results based on these traces to the results obtained with full software tracing with respect to true and false sharing invalidations as well as coherence misses.

We found that both methods reduced the number of loads captured by more than two orders of magnitude over full tracing. We discovered that both our methods have a weakness for certain programs when evaluating false-sharing, and we suggested possible solutions to resolve it. Because of trace lossiness, PMU-based tracing has larger number of false positives, in general, compared to targeted tracing. Targeted tracing allowed more control over the degree of lossiness, at the cost of much increased execution overhead.

7.4 Hardware Profile-guided Automatic Page Placement for ccNUMA

In this work, we demonstrated the use of hardware-generated filtered lossy traces for automated page placement. Our technique exploited the Itanium2 PMU to obtain low-cost whole-program traces describing a program's memory access pattern in its stable execution region (*e.g.,* a time step). This information was used to decide a good page placement.

In contrast to previous work, our technique operated completely in user space, and requires no special support from either the compiler, linker or operating system. Furthermore, it supported both static and dynamically allocated regions of memory. Finally, our technique was more *portable* because it uses *processor-centric* hardware that is available in all Itanium2 processors, in contrast to previous work that used special proprietary hardware specific to a particular platform or interconnect.

We evaluated two sources of hardware profile sources — long-latency loads and translation lookaside buffer (DTLB) misses. Overall, we showed that long-latency loads provide a better indicator for page placement than TLB misses. Page placement based on long-latency loads resulted in average wallclock time savings of more than 20% over all benchmarks, with an average one-time profiling cost of 2.7% over the overall original program wallclock time. The low overhead may make automatic automatic page placement a cheap commodity without requiring user intervention.

7.5 PFetch: Profile-guided Data Prefetching

In this work, we used memory traces to create a novel software-only prefetch scheme. Our scheme exploited predictability in the memory access stream to create prefetch predictors. Our approach unified and extended several past approaches that each targeted a different source of address predictability.

Our approach was fully automated, portable and uses novel threshold-based mechanisms for addressing prefetch accuracy, prefetch timeliness and prefetch redundancy.

We evaluated our scheme on a real machine as well as a cache simulator. We demonstrated that that significant reductions in L1D load cache misses are achievable on real

hardware with our approach. However, the performance improvements in terms of processor cycles were lower than anticipated (and, in some cases, negative). Based on experience with microbenchmarks, we reported the probable causes for this result and suggested solutions for them.

Overall, we demonstrated that memory traces represent a rich source of information about a program's behavior and can be effectively used for a wide range of performance analysis and optimization strategies.

Bibliography

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

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

[3] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, 1994.

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

[5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, June 2000.

[6] William Bolosky, Michael Scott, Robert Fitzgerald, Robert Fowler, and Alan Cox. NUMA policies and their relation to memory architecture. In *Proceedings of the fourth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 212–221, 1991.

[7] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. Proteus: A high-performance parallel-architecture simulator. In *Proceedings of the SIGMETRICS and PERFORMANCE '92 International Conference on Measurement*

and Modeling of Computer Systems, pages 247–248, New York, NY, USA, June 1992. ACM Press.

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

[9] 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*, November 2000.

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

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

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

[13] J.M. Bull and C. Johnson. Data Distribution, Migration and Replication on a cc-NUMA Architecture. In *Proceedings of the Fourth European Workshop on OpenMP*, 2002.

[14] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, University of Wisconsin, Madison, July 1996.

[15] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.

[16] M. Burtscher. Vpc3: A fast and effective trace-compression algorithm. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 167–176, N.Y., June 2004.

[17] Martin Burtscher. Vpc3 source code. http://www.csl.cornell.edu/ burtscher, 2004.

[18] Martin Christopher Carlisle. *Olden: parallelizing programs with dynamic data structures on distributed-memory machines*. PhD thesis, Princeton University, Princeton, NJ, USA, 1996.

[19] S. Chatterjee, E. Parker, P. Hanlon, and A. Lebeck. Exact analysis of the cache behavior of nested loops. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–297, June 2001.

[20] S. Chenney. Controllable and scalable simulation for animation, 2000.

[21] Trishul Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 191–202, June 2001.

[22] Trishul Chilimbi. Dynamic hot data stream prefetching for general-purpose programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 191–209, June 2002.

[23] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, May 1999.

[24] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, May 1999.

[25] Cristina Cifuentes and Mike Van Emmerik. UQBT: Adaptable binary translation at low cost. *Computer*, 33(3):60–66, March 2000.

[26] Jamison Collins, Suleyman Sair, Brad Calder, and Dean M. Tullsen. Pointer cache assisted prefetching. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 62–73, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[27] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 279–290, New York, NY, USA, 2002. ACM Press.

[28] Helen Davis, Stehpen R. Goldschmidt, and John Hennessy. Multiprocessor simulation and tracing using tango. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume II, Software, pages II–99–II–107, Boca Raton, FL, August 1991. CRC Press.

[29] 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)*, pages 292–302, December 1997.

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

[31] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2003.

[32] R. D. Falgout E. Chow, A. J. Cleary. Design of the hypre preconditioner library. In *SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, October 1998.

[33] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 133–143, New York, NY, USA, 1997. ACM Press.

[34] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

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

[36] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers. An evaluation of staged run-time optimizations in dyc. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–304, June 1999.

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

[38] Paul Havlak and Ken Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[39] J.L. Hennessy and D.A. Patterson. *Computer Organization and Design*. Morgan Kaufmann, 2nd edition, 1997.

[40] J. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millenium. *IEEE Computer*, 33(7):28–35, July 2000.

[41] Hewlett-Packard. *Perfmon project*.

[42] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. Informing memory operations: Providing memory performance feedback in modern processors. In *International Symposium on Computer Architecure*, pages 260–270, May 1996.

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

[44] Ibrahim Hur and Calvin Lin. Memory prefetching using adaptive stream detection. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 397–408, Washington, DC, USA, 2006. IEEE Computer Society.

[45] Tatsushi Inagaki, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Stride prefetching by dynamically inspecting objects. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 269–277, New York, NY, USA, 2003. ACM Press.

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

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

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

[49] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 252–263, New York, NY, USA, 1997. ACM Press.

[50] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 364–373, New York, NY, USA, 1990. ACM Press.

[51] Dongkeun Kim, Steve Shih wei Liao, Perry H. Wang, Juan del Cuvillo, Xinmin Tian, Xiang Zou, Hong Wang, Donald Yeung, Milind Girkar, and John P. Shen. Physical experimentation with prefetching helper threads on intel's hyper-threaded processors. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 27, Washington, DC, USA, 2004. IEEE Computer Society.

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

[53] J. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software Practice & Experience*, 24(2):197–218, February 1994.

[54] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, June 1995.

[55] Alvin R. Lebeck and David A. Wood. Cache profiling and the SPEC benchmarks: A case study. *Computer*, 27(10):15–26, October 1994.

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

[57] Mikko H. Lipasti, William J. Schmidt, Steven R. Kunkel, and Robert R. Roediger. Spaid: software prefetching in pointer- and call-intensive environments. In *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 231–236, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[58] Jiwei Lu, Abhinav Das, Wei-Chung Hsu, Khoa Nguyen, and Santosh G. Abraham. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 93–104, Washington, DC, USA, 2005. IEEE Computer Society.

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

[60] Chi-Keung Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. Ispike: A post-link optimizer for the intel®itanium®architecture. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 15, Washington, DC, USA, 2004. IEEE Computer Society.

[61] Neville Manning. Sequitur source code. http://sequence.rutgers.edu/sequitur, 2005.

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

[63] J. Marathe and F. Mueller. Detecting memory performance bottlenecks via binary rewriting. In *Workshop on Binary Translation*, September 2002.

[64] J. Marathe and F. Mueller. Hardware profile-guided automatic page placement for ccnuma systems. In *PPOPP*, pages 90–99, March 2006.

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

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

[67] Jaydeep Marathe, Frank Mueller, and Bronis R. de Supinski. A hybrid hardware/software approach to efficiently determine cache coherence bottlenecks. In *International Conference on Supercomputing*, June 2005.

[68] 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*, page (to appear), 2004.

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

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

[71] Tushar Mohan, Bronis R. de Supinski, Sally A. McKee, Frank Mueller, Andy Yoo, and Martin Schulz. Identifying and exploiting spatial regularity in data memory references. In *Supercomputing*, November 2003.

[72] T. C. Mowry and C.-K. Luk. Predicting data cache misses in non-numeric applications through correlation profiling. In *MICRO-30*, pages 314–320, December 1997.

[73] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. *SIGPLAN Notices*, 27(9):62–73, 1992.

[74] F. Mueller, T. Mohan, B. R. de Supinski, S. A. McKee, and A. Yoo. Partial data traces: Efficient generation and representation. In *Workshop on Binary Translation*, IEEE Technical Committee on Computer Architecture Newsletter, October 2001.

[75] A. Nagarajan. *Analyzing Memory Performance Bottlenecks in OpenMP Programs on SMP Architectures using ccSIM.* PhD thesis, North Carolina State University, July 2003.

[76] Kyle J. Nesbit and James E. Smith. Data cache prefetching using a global history buffer. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 96, Washington, DC, USA, 2004. IEEE Computer Society.

[77] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Proc. of the 3rd Workshop on Runtime Verification*, July 2003.

[78] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2/3):103–??, ???? 1997.

[79] Craig G. Nevill-Manning and Ian H. Witten. Linear-time, incremental hierarchy inference for compression. In *Data Compression Conference*, pages 3–11, 1997.

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

[81] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta, and E. Ayguade. User-level dynamic page migration for multiprogrammed shared-memory multiprocessors. In *International Conference on Parallel Programming*, pages 95–103, August 2000.

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

[83] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesus Labarta, and Eduard Ayguade. UPMLIB: A runtime system for tuning the memory performance of openmp programs on scalable shared-memory multiprocessors. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 85–99, 2000.

[84] Lise Noordergraaf and Robert Zak. Performance experiences on Sun's Wildfire prototype. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, 1999.

[85] Markus F.X.J. Oberhumer. LZO real-time data compression library, 2002.

[86] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *ISCA '94: Proceedings of the 21ST annual international symposium on Computer architecture*, pages 24–33, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

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

[88] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The SimOS approach. *IEEE parallel and distributed technology: systems and applications*, 3(4):34–43, Winter 1995.

[89] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 115–126, New York, NY, USA, 1998. ACM Press.

[90] Amir Roth and Gurindar S. Sohi. Effective jump-pointer prefetching for linked data structures. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 111–121, Washington, DC, USA, 1999. IEEE Computer Society.

[91] Suleyman Sair, Timothy Sherwood, and Brad Calder. Quantifying load stream behavior. In *HPCA*, pages 197–, 2002.

[92] Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano, and Yoshio Tanaka. Design of OpenMP compiler for an SMP cluster. In *EWOMP '99*, pages 32–39, September 1999.

[93] Shigehisa Satoh, Kazuhiro Kusano, and Mitsuhisa Sato. Compiler optimization techniques for openMP programs. *Scientific Programming*, 9(2-3):131–142, 2001.

[94] J. Seward. Libbzip2 source code. http://www.bzip.org/index.html, 2005.

[95] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[96] J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 588–597, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

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

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

[99] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[100] J. Tao, M. Schulz, and W. Karl. A simulation tool for evaluating shared memory performance. In *Proc. of the 36th Annual Simulation Symposium*, April 2003.

[101] J. Tao and J. Weidendorfer. Cache simulation based on runtime instrumentation for OpenMP applications. In *Proc. of the 37th Annual Simulation Symposium*, pages 97–103, April 2004.

[102] J. M. Tendler, J. S. Dodson, J. S. Fields, Jr., H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, January 2002.

[103] Christian Thiffault, Michael Voss, Steven T. Healey, and Seon Wook Kim. Dynamic instrumentation of large-scale mpi/openmp applications. In *International Parallel and Distributed Processing Symposium*, April 2003.

[104] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Using hardware counters to automatically improve memory performance. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 46, Washington, DC, USA, 2004. IEEE Computer Society.

[105] D. Ung and C. Cifuentes. Optimising hot paths in a dynamic binary translator. In *Workshop on Binary Translation*, October 2000.

[106] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on ccNUMA compute servers. In *Proceedings*

of the seventh International Conference on Architectural Support for Programming Languages and Operating Systems, pages 279–289, 1996.

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

[108] J. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *Journal of Parallel Distributed Computing*, 63(9):853–865, September 2003.

[109] D.A.B. Weikle, S.A. McKee, Kevin Skadron, and Wm.A. Wulf. Caches as filters: A framework for the analysis of caching systems. In *Grace Murray Hopper Conference*, September 2000.

[110] M. E. Wolf and M. S. Lam. A data locality optimizating algorithm. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[111] Youfeng Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 210–221, New York, NY, USA, 2002. ACM Press.

[112] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, 1995.

[113] www.openmp.org. *Official OpenMP Specification*, May 2005.

[114] Zheng Zhang and Josep Torrellas. Speeding up irregular applications in shared-memory multiprocessors: memory binding and group prefetching. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 188–199, New York, NY, USA, 1995. ACM Press.

[115] Zheng Zhang and Josep Torrellas. Speeding up irregular applications in shared-memory multiprocessors: memory binding and group prefetching. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 188–199, New York, NY, USA, 1995. ACM Press.

[116] Yutao Zhong, Maksim Orlovich, Xipeng Shen, and Chen Ding. Array regrouping and structure splitting using whole-program reference affinity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.

[117] Huiyang Zhou and Thomas M. Conte. Enhancing memory-level parallelism via recovery-free value prediction. *IEEE Trans. Comput.*, 54(7):897–912, 2005.