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

JAYDEEP MARATHE, FRANK MUELLER
North Carolina State University
TUSHAR MOHAN
IBM India Research Lab
SALLY A. MCKEE
Cornell University
and
BRONIS R. DE SUPINSKI and ANDY YOO
Lawrence Livermore National Laboratory

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

A preliminary version of this article appeared in the *International Symposium on Code Generation and Optimization, 2003* [Marathe et al. 2003]. F. Mueller was supported in part by the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under subcontracts Nos. B518219 and B540203, NFS CAREER CCR-0237570, CNS-0406305, CCF-0429653; S. A. McKee was supported in part by LLNL LDRD 01-ERD-043, NSF CCR-0073532; B.R. de Supinkski and A Yoo were supported in part to perform this research under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under Contract W-7405-Eng-48, UCRL-JRNL-22545. This research was supported in part by the National Science Foundation through the San Diego Supercomputer Center under Grant CCR-0237570 using the DataStar computing system. T. Mohan's contribution was in the course of his thesis at the University of Utah. This article does not necessarily reflect or represent the views of IBM, Inc.

Author's addresses: J. Marathe, F. Mueller (Contact Author), Department of Computer Science, North Carolina State University, Raleigh, NC 27695-7534; email: mueller@cs.ncsu.edu; T. Mohan, IBM India Research Lab, Block I, IIT, Hauz Khas, New Delhi 110016, India; S. A. McKee, School of Electrical and Computer Engineering, Cornell University, Ithaca, NY 14853; B. R. de Supinski, A. Yoo, Lawrence Livermore National Laboratory, Center for Applied Scientific Computing, L-557, Livermore, CA 94551.

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.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers; optimization*; B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Dynamic binary rewriting, program instrumentation, data trace generation, data trace compression, cache analysis

ACM Reference Format:

Marathe, J., Mueller, F., Mohan, T., McKee, S. A., de Supinski, B. R., and Yoo, A. 2007 METRIC: Memory tracing via dynamic binary rewriting to identify cache inefficiencies. ACM Trans. Program. Lang. Syst. 29, 2, Article 12 (April 2007), 36 Pages. DOI=10.1145/1216374.1216380 http://doi.acm.org/10.1145/1216374.1216380.

1. 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 the *causes* of potential memory access inefficiencies. In this article, 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 datasets 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 [Vetter and Mueller 2003]. 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 [Mueller et al. 2001] and binary rewriting [Marathe and Mueller 2002] to detect memory hierarchy bottlenecks. METRIC is also influenced by our work with large-scale benchmarks [Vetter and Mueller 2003], another example of data-centric computation where data sizes exceed cache capacities.

In this article, 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. [2001]) 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 [Buck and Hollingsworth 2000a] to exploit *dynamic binary rewriting*, or postlink-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 the requirements of recompiling or relinking.

Dynamic binary rewriting can capture memory references of the entire application, including library routines, and works equally well for the mixed language applications commonly found in production of scientific codes [Vetter and Mueller 2003]. The techniques can be adapted to address changing input dependencies and application modes, namely, 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 [Bala et al. 2000].

2. THE METRIC FRAMEWORK

The METRIC framework, shown in Figure 1, uses partial access traces for memory hierarchy simulation. Our framework extracts these comparatively small, low-overhead access traces without compiler or linker support, that is, traces can be extracted from arbitrary 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

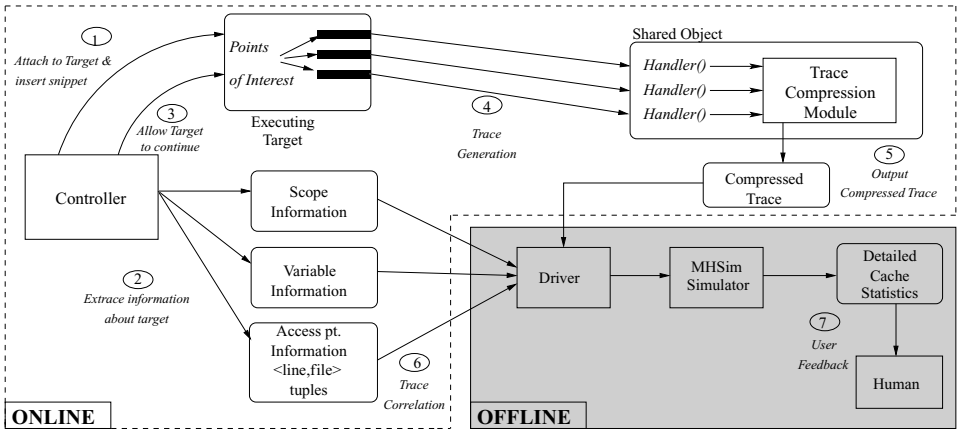

Fig. 1. The METRIC framework.

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 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 (e.g., 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, namely, 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 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 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. In particular, IBM's AIX and Intel/KAI compilers for the PowerPC do not suffer in their optimization levels when debugging information is retained. While some debugging information may suffer in accuracy due to certain optimizations, memory references are usually not affected. Thus, compiling with symbolic information only increases executable size, without significant performance degradation.

3. 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 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. The access stream to be compressed consists of individual records described by the tuple <*point_id, EA*>. Moreover, `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 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.

3.1 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

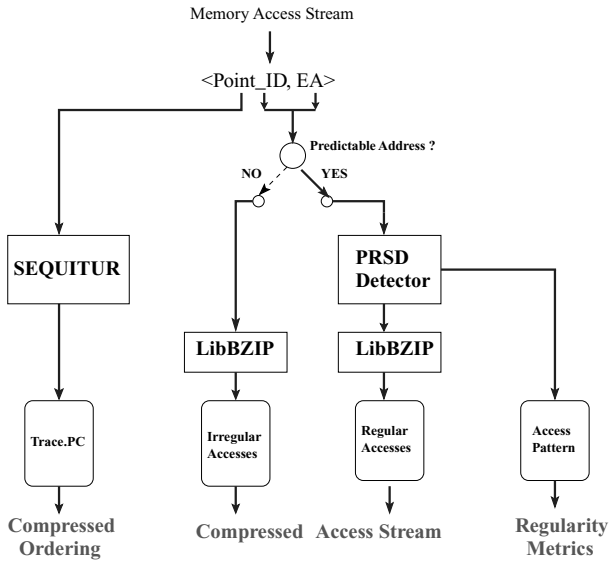

Fig. 2. Overall compression algorithm.

memory references. SEQUITUR is described by Nevill-Manning and Witten [1997a]. It converts a trace of symbols into a context-free grammar, and has time overhead linear in the number of symbols in the trace [Nevill-Manning and Witten 1997b]. 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*, that is, compressed traces can be used directly for cache simulation, without an intermediate trace expansion step.

3.2 Compressing Trace Accesses

The accesses generated by each access point, namely, 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 [1991]. 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, and 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

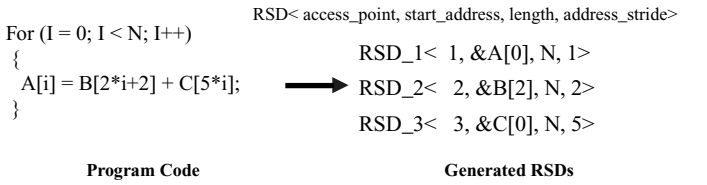

Fig. 3. Example RSDs.

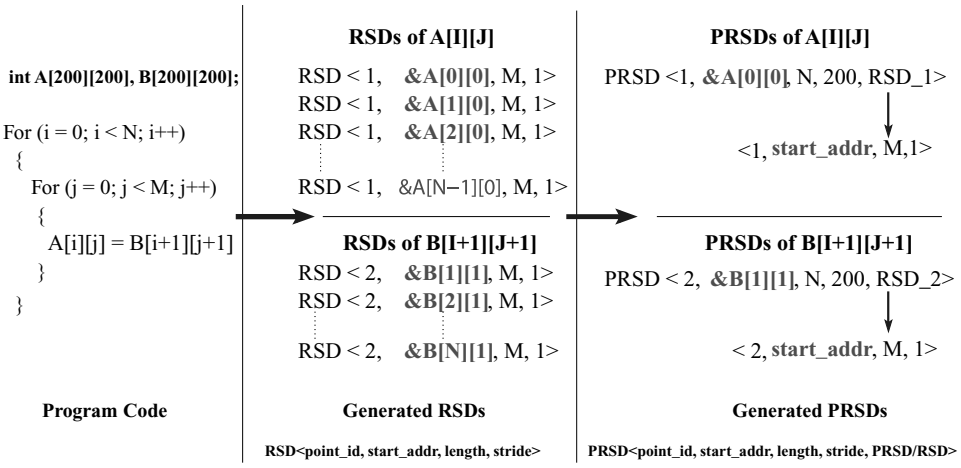

Fig. 4. Example PRSDs.

RSDs may be an arbitrary function. We restrict ourselves to constants in this article, since we require fast online techniques to recognize RSDs. In different contexts, we 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 3, assuming that 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 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 4, assuming the size of integers is one and that 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 the 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 [Seward 2005]. BZIP2 compresses using a block sorting algorithm described by Burrows and Wheeler [1994].

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 to achieve a deeper understanding of the program's memory access behavior.

4. 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. By contrast, RSDs have height zero (since they themselves do not have child RSDs).

The algorithm is intuitive. It 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 triggers formation of a new PRSD, and potentially *flushes* the current PRSD to the output buffer.

4.1 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, namely, PRSDs with increasing heights. The current data access to be compressed is processed at the lowest level, that is, 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

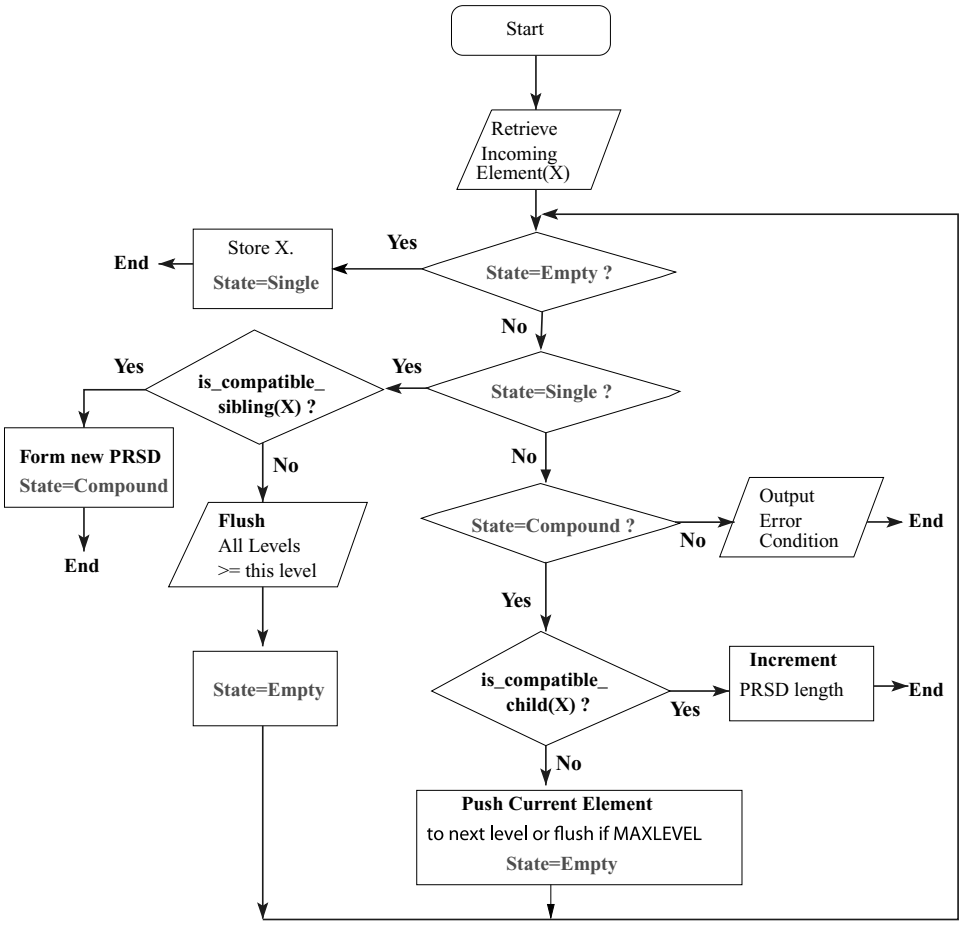

Fig. 5. PRSD detector flowchart: processing in a level.

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.

4.2 Per-Level Processing

Figure 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 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 and length, and their children are 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 same start_address as the start_address of Y and an address_stride of the difference between the start_addresses of Y and X, and will encapsulate Y as the child_prsd.

If X and Y are not compatible siblings, a change in the data access pattern is detected, for example, 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 whether 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 whether 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, that is, 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. Specifically, 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 reprocessed at a higher level.

4.3 Example

Figure 6 shows the operation of the PRSD detection algorithm for the A[i][j] reference shown in Figure 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.

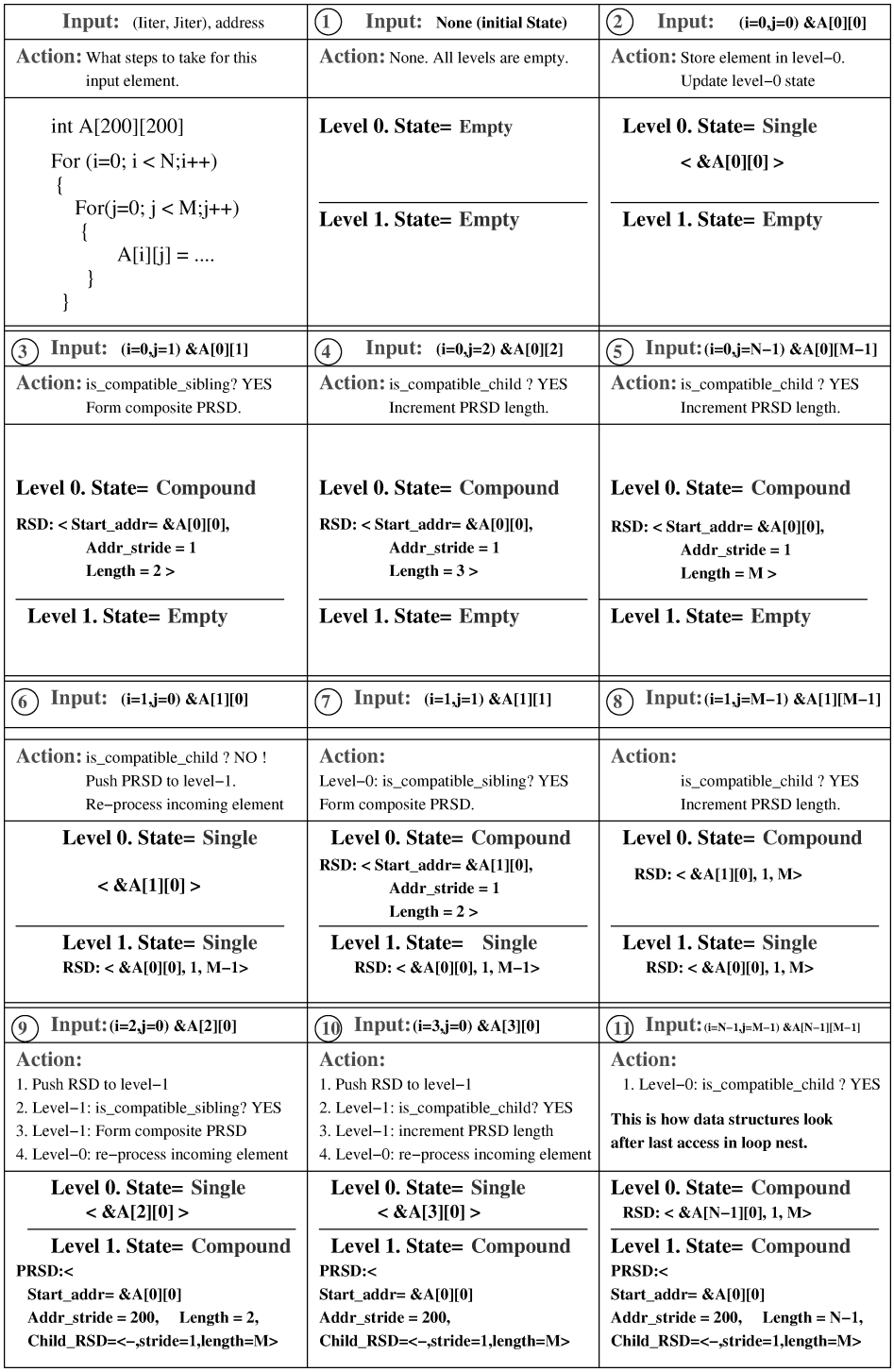

Fig. 6. PRSD detection example.

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 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 that `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]), that is, `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 current 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 preresident 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 preresident PRSD. This will cause the length of the preresident 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 current 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 preresident PRSD.

4.4 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). Here, `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.

4.5 Time Complexity

Since we must look at each incoming element so as 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 reprocessed. The number of reprocessing 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.

5. 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 state-of-the-art compression algorithm based on using value predictors for data compression [Burtscher 2004a].

5.1 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

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

Table I. 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	

the second-stage compressor. In our experiments, we use the VPC3 source code obtained from the author's website [Burtscher 2004b] and couple the output to a second-stage compressor based on BZIP2 [Seward 2005].

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 a more effective compression rate for most benchmarks, compared to several contemporary compression algorithms (SEQUITUR, BZIP2, GZIP) [Burtscher 2004a]. 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 9). These metrics, along with the results generated by the simulator, provide insight into the application's memory access behavior.

5.2 Experimental Setup

For our compression scheme we used the open source implementation of SEQUITUR [Manning 2005]. All benchmarks were compiled at -O2 optimization level on an IBM POWER4 platform. All benchmarks used "training" datasets. The static call graph of the target program was traversed with `main` as the root, and all memory access points in the call graph were instrumented. Up to one billion (10^9) accesses were traced and compressed online for each benchmark. All benchmarks reached the one billion limit, except for 177.mesa (8×10^6 total accesses) and 188.ammp (531×10^6 total accesses).

5.3 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 65,536 memory access points. Thus the

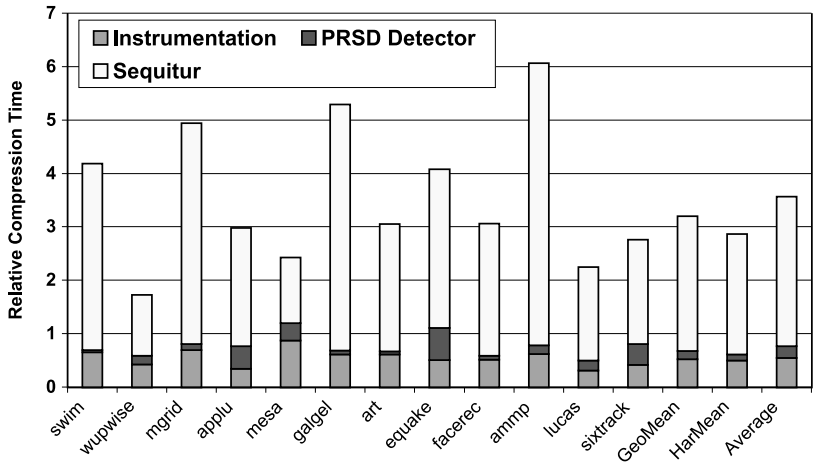

Fig. 7. Execution time breakup for our compression scheme, relative to VPC3 execution time.

total uncompressed trace size is (# total_records) * 6. The compression rate is calculated as $\frac{size\ of\ un\text{-}compressed\ trace}{size\ of\ compressed\ trace}$.

Table I 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 about 25% greater than the value for VPC3.

5.4 Comparison of Compression Times

Figure 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 compression 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 9) 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.

6. 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` [Mellor-Crummey et al. 2001]. 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 to follow. 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 subexpression 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 [Vetter and Mueller 2003]. 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 framework 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 only limited to the duration of monitoring.

7. 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.

8. MHSIM-GENERATED METRICS

MHSim generates metrics for each level of cache and also for the data TLB. Metrics can be aggregated by reference, variable, and loop nest. We shall list and describe each metric and later discuss its 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 nontemporal hits. Temporal hits include hits caused by both *self-reuse* (the same reference point accesses a memory location multiple times) and *cross-reuse* (different reference points access the same memory location).

—*Spatial hit fraction*: This is defined as a 1 - temporal_hit_fraction, namely, nontemporal hits are classified as purely spatial hits.

—*Spatial reuse*: This value gives the average fraction of the memory line (in bytes) that was *used*, that is, 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.

9. 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*: This is 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 [Mohan et al. 2003]. For example, the predictable nature of the access point can be exploited by prefetching, which caches future data access early so as to lessen effective access latencies.

—*Mean stream length*: The average of the length of all RSDs generated at this point.

—*Number of distinct lengths*: Number of distinct RSD lengths seen at this access point.

—*Percentage of the distribution of distinct lengths*: The distribution of RSDs according to their lengths.

—*Number of distinct strides*: Number of address strides for all RSDs seen at this point.

—*Percentage of 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 [Mohan et al. 2003]. 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, that is, 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. This 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 they 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.
Percentage 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).

Fig. 8. Use of metrics for performance diagnosis.

10. 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 underlying causes of memory access inefficiencies? Figure 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 [Marathe et al. 2003]. This article 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.

10.1 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=256KB, associativity=8, line size=128, writeback cache, LRU replacement policy. This configuration is similar to the L2 cache of the Itanium2 processor [Intel 2004]. The per-reference results generated by the simulator are shown in Figure 9.

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

Fig. 9. Original per-reference memory usage statistics.

Figure 9(a) shows the cache metrics generated by the simulator, and Figure 9(b) shows the stream metrics generated by the PRSD detector.

10.1.1 *Analysis.* The reference name shown in the results has the following syntax: *VariableName_Accesstype_id*. Here, *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 article.

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 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`

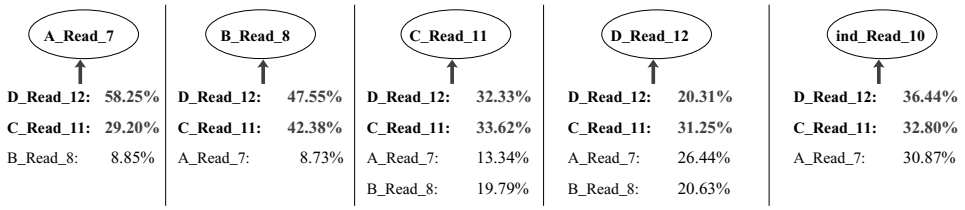

Fig. 10. Evictors for each reference.

bring data into the cache that is not reused (as indicated by their low spatial reuse values) and evict a significant amount of preresident data from the cache (as indicated by the perreference 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.

10.1.2 *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 patterns. Instead of trying to reorder their access patterns, we can try to reduce their detrimental impact on the cache by asking the memory syste *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 [Intel 2004]. 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 preresident 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 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 next:

File	Line	Reference	SourceRef	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**

Fig. 11. Optimized per-reference memory usage statistics.

```
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 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 less than 3% in the original results. This indicates that we are now realizing *intertimestep locality*—the data is brought into the cache during the first timestep and almost always remains in cache until it is accessed again during the next timestep.

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 programs. The values for the two runs are shown in Figure 12. The number of L2 misses reduces from 42,214 in the original program to 32,072 in the optimized version (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

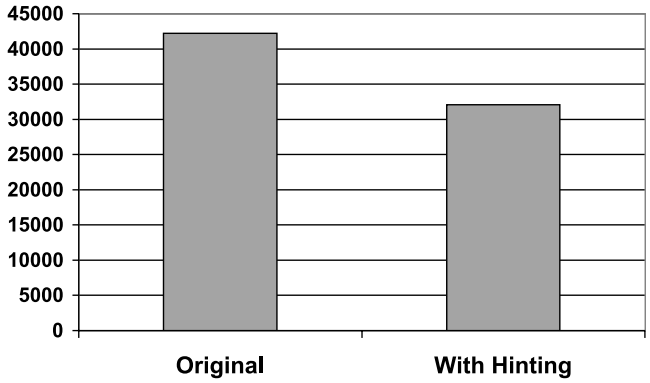

Fig. 12. Comparison of L2 cache misses.

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, reuse hinting may actually be detrimental). The compilers evaluated (Intel icc 8.0, gcc 3.4) did not automatically set the nontemporal hints for the `D_Read` and `C_Read` (for the optimized code, we inserted the hints manually using inline assembly functions).

10.2 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=32KB, 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

[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**	Average 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

Fig. 13. Original per-reference memory usage statistics.

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 relatively low spatial reuse value (**17.8%**). The per-reference results are shown in Figure 13. Figure 13(a) shows the metrics generated by the cache simulator, and 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.

10.2.1 *Analysis*. `B_Read_3` has the worst possible cache performance, all of its accesses are misses. 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 (8,000), 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`, however, `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 results 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 (8,000) seen for `B_Read_3`.

10.2.2 *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 [Tendler et al. 2002]. 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 8,000). 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**

Fig. 14. Performance of original and optimized programs.

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 optimized programs. The results are shown in Figure 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 targeting 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.

10.3 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,

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

Fig. 15. Original per-reference memory usage statistics.

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=128KB, 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 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 15. Figure 15(a) shows the cache metrics generated by the simulator, while 15(b) shows the per-reference stream metrics generated by the PRSD detector during trace compression.

10.3.1 *Analysis.* The per-reference results for all references show 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 highly predictable accesses, with a regularity ratio of 1.0 and long average lengths (8,192). Most crucially, each reference 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, which removed data accessed by this reference from the cache. The list of evictors is

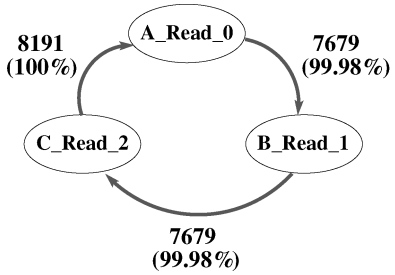

Fig. 16. Evictor graph.

shown graphically in Figure 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, that is, 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 can be fully used, which explains the low spatial reuse values.

10.3.2 *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 next:

```
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**

Fig. 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 an 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 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 runtime). Thus, we need tools like METRIC to analyze such scenarios.

11. RELATED WORK

Regular section descriptors represent a particular instance of a common concept in memory optimizations, either in software or hardware. For instance, RSDs [Havlak and Kennedy 1991] are virtually identical to the *stream descriptors* used at about the same time in the compiler and memory systems work inspired by WM architecture [Wulf 1992].

Atom has been widely used as a binary rewriting tool to statically insert instrumentation code into application binaries [Srivastava and Eustace 1994]. Dynamic binary rewriting enhances this approach through its ability to select place and time for instrumentation dynamically. This allows the generation of partial address traces, for example, for frequently executed regions of code and a limited number of iterations with a code section. In addition, DynInst makes dynamic binary rewriting a portable approach.

Weikle et al. [2000] describe an analytic framework for evaluating caching systems. Their approach views caches as filters, and one component of the framework is a trace specification notation called *TSpec*. TSpec is similar to the RSDs described here in that it provides a more formal mechanism by which researchers may communicate with clarity about the memory references generated by a processor. The TSpec notation is more complex than RSDs, since it is also the object on which the cache filter operates.

Buck and Hollingsworth performed a simulation study to pinpoint the hot spots of cache misses based on hardware support for data trace generation [2000b]. Hardware counter support, in conjunction with interrupt support on overflow for a cache miss counter, was compared to miss counting in selected memory regions. The former approach is based on probing to capture data misses at a certain frequency (e.g., one out of 50,000 misses). The latter approach performs a binary (or n-way) search over the data space to identify the location of the most frequently occurring misses. Sampling was reported to yield less accurate results than searching. The approach based on searching provided accurate results (mostly less than 2% error) for these simulations. Unfortunately, hardware support for these two approaches is not yet readily available (with the exception of the IA64) or there is a lack of documentation for this support (as confirmed by one vendor). In addition, interrupts on overflow are imprecise due to instruction-level parallelism. The data reference causing an interrupt is only known to be located in "close vicinity" to the interrupted instruction, which complicates the analysis. In contrast, our approach to generating traces is applicable to present-day architectures, is portable and precise in locating data references, and does not require the overhead of interrupt handling. Other approaches to determining the causes of cache misses, such as informing memory operations, are also based on hardware support and presently not supported in contemporary architectures [Horowitz et al. 1996; Mowry and Luk 1997].

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 [Mellor-Crummey et al. 2001]. 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. The 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 [2001]. This approach shares its goal of cache correlation with our work. CProf [Lebeck and Wood 1994] is a similar tool that relies on postlink-time binary editing through EEL [Larus and Ball 1994; Larus and

Schnarr 1995] 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 [1997]. Our work differs in the fundamental approach of rewriting binaries, which neither is restricted to a special compiler or programming language nor precludes the analysis of library routines. Another major difference addresses the overhead of the 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 [1999a; 1999b] as well as quantitative metrics to assess memory bottlenecks within the data reference stream [Chilimbi 2001]. 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 than a WPS with its need for states and transitions. Ding and Zhong et al. [2003] predict program locality from profiles using the approximate reuse distance of referenced data to identify regular and irregular reference patterns. 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 [Zhong et al. 2004]. 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 [Ghosh et al. 1999] or symbolic reference analysis at the source level based on Presburger formulas [Chatterjee et al. 2001]. 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 [Sites et al. 1993; Bala et al. 2000; Cifuentes and Emmerik 2000; Ung and Cifuentes 2000; Grant et al. 1999]. The main thrust of these techniques is program transformation based on knowledge about taken execution paths, such as trace scheduling. 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 [DeRose et al. 2002]. 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 [Mohan et al. 2003]. 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 memory access behavior. More importantly, our current work supplements stream metrics with cache usage metrics (per-reference statistics, evictor information). The richer information about 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 [Marathe et al. 2004]. 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 [Marathe et al. 2005]. This article, in contrast, concentrates on application-level characterization of *uniprocessor* memory hierarchy metrics.

12. CONCLUSION

In this article, 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 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.

REFERENCES

BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1–12.

BUCK, B. AND HOLLINGSWORTH, J. 2000a. An API for runtime code patching. *Int. J. High Perform. Comput. Appl. 14,* 4, 317–329.

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

BURROWS, M. AND WHEELER, D. J. 1994. A block-sorting lossless data compression algorithm. Tech. Rep. 124.

BURTSCHER, M. 2004a. Vpc3: A fast and effective trace-compression algorithm. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (New York). 167–176.

BURTSCHER, M. 2004b. Vpc3 source code. http://www.csl.cornell.edu/ burtscher/research/tracecompression/.

CHATTERJEE, S., PARKER, E., HANLON, P., AND LEBECK, A. 2001. Exact analysis of the cache behavior of nested loops. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 286–297.

CHILIMBI, T. 2001. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 191–202.

CHILIMBI, T., DAVIDSON, B., AND LARUS, J. 1999. Cache-Conscious structure definition. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 13–24.

CHILIMBI, T., HILL, M., AND LARUS, J. 1999b. Cache-Conscious structure layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1–12.

CIFUENTES, C. AND EMMERIK, M. 2000. UQBT: Adaptable binary translation at low cost. *Comput. 33,* 3 (Mar.), 60–66.

DEROSE, L., EKANADHAM, K., HOLLINGSWORTH, J. K., AND SBARAGLIA, S. 2002. SIGMA: A simulator infrastructure to guide memory analysis. In *Proceedings of the ACM/IEEE SC Conference*.

DING, C. AND ZHONG, Y. 2003. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

GHOSH, S., MARTONOSI, M., AND MALIK, S. 1999. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst. 21,* 4, 703–746.

GRANT, B., PHILIPOSE, M., MOCK, M., CHAMBERS, C., AND EGGERS, S. 1999. An evaluation of staged run-time optimizations in dyc. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 293–304.

HAVLAK, P. AND KENNEDY, K. 1991. An implementation of interprocedural bounded regular section analysis. *IEEE Trans. Parallel Distrib. Syst. 2,* 3 (Jul.), 350–360.

HOROWITZ, M., MARTONOSI, M., MOWRY, T., AND SMITH, M. 1996. Informing memory operations: Providing memory performance feedback in modern processors. In *Proceedings of the International Symposium on Computer Architecure*, 260–270.

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

LARUS, J. AND BALL, T. 1994. Rewriting executable files to measure program behavior. *Softw. Pract. Experi. 24,* 2 (Feb.), 197–218.

LARUS, J. AND SCHNARR, E. 1995. EEL: Machine-Independent executable editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 291–300.

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

LEBECK, A. AND WOOD, D. 1997. Active memory: A new abstraction for memory system simulation. *ACM Trans. Model. Comput. Simul. 7,* 1 (Jan.), 42–77.

MANNING, N. 2005. Sequitur source code. http://sequence.rutgers.edu/sequitur/sequitur.cc.

MARATHE, J. AND MUELLER, F. 2002. Detecting memory performance bottlenecks via binary rewriting. In *Proceedings of the Workshop on Binary Translation*.

MARATHE, J., MUELLER, F., AND DE SUPINSKI, B. R. 2005. A hybrid hardware/software approach to efficiently determine cache coherence bottlenecks. In *International Conference on Supercomputing*. accepted.

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

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

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

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

MOWRY, T. AND LUK, C.-K. 1997. Predicting data cache misses in non-numeric applications through correlation profiling. In *MICRO-30*, 314–320.

MUELLER, F., MOHAN, T., DE SUPINSKI, B. R., MCKEE, S. A., AND YOO, A. 2001. Partial data traces: Efficient generation and representation. In *Workshop on Binary Translation*. IEEE Technical Committee on Computer Architecture Newsletter.

NEVILL-MANNING, C. G. AND WITTEN, I. H. 1997a. Compression and explanation using hierarchical grammars. *Comput. J. 40,* 2–3.

NEVILL-MANNING, C. G. AND WITTEN, I. H. 1997b. Linear-Time, incremental hierarchy inference for compression. In *Proceedings of the Data Compression Conference*, 3–11.

SEWARD, J. 2005. Libbzip2 source code. http://www.bzip.org/index.html.

SITES, R., CHERNOFF, A., KIRK, M., MARKS, M., AND ROBINSON, S. 1993. Binary translation. *Commun. ACM 36,* 2 (Feb.), 69–81.

SRIVASTAVA, A. AND EUSTACE, A. 1994. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 196–205.

TENDLER, J. M., DODSON, J. S., FIELDS, JR., J. S., LE, H., AND SINHAROY, B. 2002. POWER4 system microarchitecture. *IBM J. Res. Develop. 46,* 1 (Jan.), 5–25.

UNG, D. AND CIFUENTES, C. 2000. Optimising hot paths in a dynamic binary translator. In *Proceedings of the Workshop on Binary Translation*.

VETTER, J. AND MUELLER, F. 2003. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *J. Parallel Distrib. Comput. 63,* 9 (Sept.), 853–865.

WEIKLE, D., MCKEE, S. A., SKADRON, K., AND WULF, W. 2000. Caches as filters: A framework for the analysis of caching systems. In *Proceedings of the Grace Murray Hopper Conference*.

WULF, W. 1992. Evaluation of the WM architecture. In *Proceedings of the International Symposium on Computer Architecture*, 382–390.

ZHONG, Y., ORLOVICH, M., SHEN, X., AND DING, C. 2004. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Received May 2005; revised August 2006; accepted September 2006