METRIC: Tracking Down Inefficiencies in the Memory Hierarchy via Binary Rewriting *

Jaydeep Marathe[1], Frank Mueller[1], Tushar Mohan[2],
Bronis R. de Supinski[4], Sally A. McKee[3], Andy Yoo[4]

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

[2] School of Computing
University of Utah
Salt Lake City, UT 84112

[3] School of ECE
Cornell University
Ithaca, NY 14853

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

mueller@cs.ncsu.edu, phone: (919) 515-7889

Abstract

In this paper, we present METRIC, an environment for determining memory inefficiencies by examining data traces. METRIC is designed to alter the performance behavior of applications that are mostly constrained by their latency to resolve memory references. We make four primary contributions in this paper. First, we present methods to extract partial data traces from running applications by observing their memory behavior via dynamic binary rewriting. Second, we present a methodology to represent partial data traces in constant space for regular references through a novel technique for online compression of reference streams. Third, we employ offline cache simulation to derive indications about memory performance bottlenecks from partial data traces. By exploiting summarized memory metrics, by-reference metrics as well as cache evictor information, we can pin-point the sources of performance problems. Fourth, we demonstrate the ability to derive opportunities for optimizations and assess their benefits in several experiments resulting in up to 40% lower miss ratios.

1. Introduction

Today, computing speed is often bound by the data path, *i.e.*, the ability of the memory hierarchy to deliver data in time to the processor. Contemporary architectures experience as much as 50% stall cycles for repetitive data-centric tasks, in the case of server workloads even up to 70% stalls [25]. Furthermore, processor speeds increase at a rate of approximately 60% per year while memory latencies are reduced by only 7% per year resulting in an increasing gap between processor speeds and memory latencies. Thus, locating and eliminating sources of inefficiencies in the memory hierarchy can potentially impact application performance to a significant degree.

Incremental memory hierarchy simulation by capturing the address trace of an application is a highly accurate method of isolating problems in the memory hierarchy. However, a significant problem with this method is the prohibitive overhead of computation and stable storage size requirements associated with capturing the *complete* address trace of the target, which could potentially consist of millions of accesses. *Partial* data traces represent a subset of the access footprint of the target and may be comparatively small and less expensive to collect, allowing selective capture of the most critical data access points in the target.

The objective of this work is to illustrate the use of partial data traces for incremental memory hierarchy simulation, a central component of METRIC (MEmory TRacIng without re-Compiling), a tool we developed to detect memory hierarchy bottlenecks drawing upon our previous experience with partial data traces [24] and binary rewriting [21]. It is also influenced by our work with large-scale benchmarks [30], another example of data-centric computation where the data sizes exceed cache capacities.

METRIC exploits *dynamic binary rewriting* by building on the instrumentation framework DynInst [2]. Dynamic binary rewriting refers to the post-link time manipulation of binary executables, potentially allowing program transformation even while the target is executing. This approach is superior to conventional instrumentation, which generally requires compiler interaction (*'e.g.*, for profiling) or the inclusion of special libraries (*e.g.*, for heap monitoring), since it obviates the requirements of recompiling or relinking. We also contribute a cache analysis approach, based on prior work [22], that lets us process these partial data traces and results not only in summary information, such as miss ratios, but also reports detailed evictor information for source-related data structures.

*This work was supported in part through the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under subcontracts # B518219 (Mueller) and LLNL LDRD 01-ERD-043 (McKee), NSF CCR award 0073532 (McKee) and was performed in part nder the auspices of the U. S. Department of Energy by University of California Lawrence Livermore National Laboratory under Contract W-7405-Eng-48, UCRL-JC-151497 (de Supinski/Yoo).

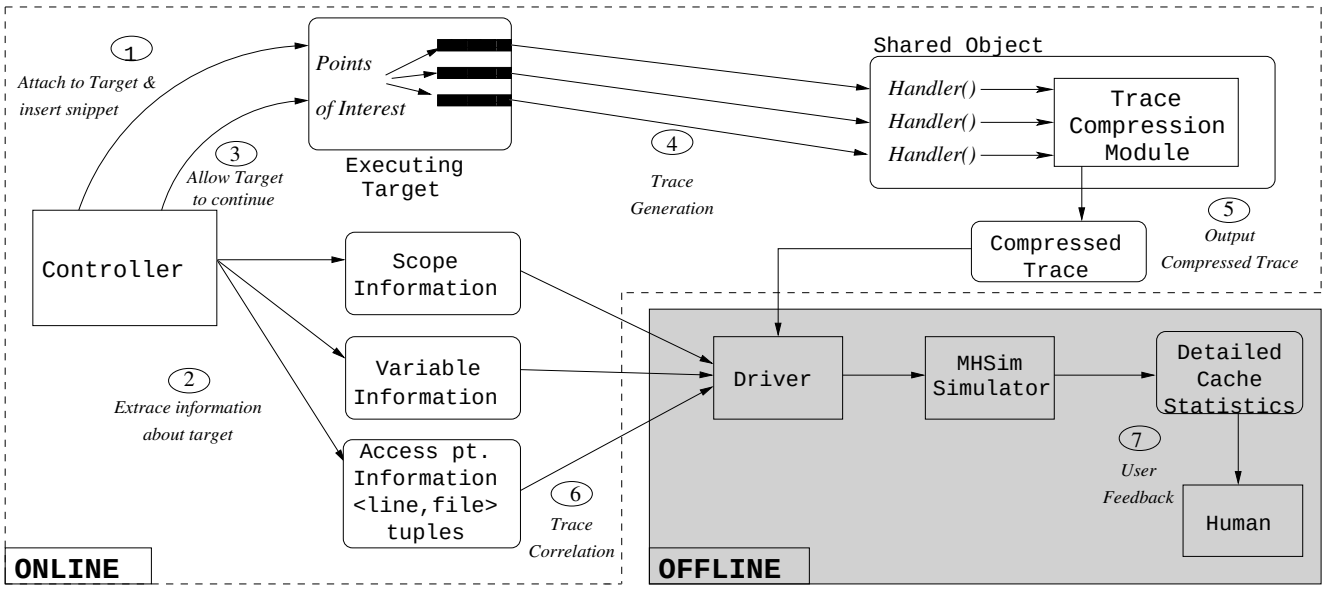


Figure 1. The METRIC Framework

The advantage of dynamic binary rewriting is its ability to capture memory references of the entire application, including library routines and mixed-language application, such as commonly found in scientific production codes [30]. Another motivation is its ability to cater for input dependencies and application modes, *i.e.*, changes over time in application behavior. This work is also influenced by findings that binary manipulation techniques offer new opportunities for program transformations, which have been shown to potentially yield performance gains beyond the scope of static code optimization without profile-guided feedback [1].

The paper is structured as follows. First, the METRIC framework is introduced. Next, the generation, representation and compression of partial data traces is discussed in detail. Then, incremental cache simulation for these partial data traces is presented and metrics to assess memory throughput are discussed. Finally, we reflect on related work, discuss future research directions and summarize.

2. The METRIC Framework

One of the central objectives of our work is to capture the memory behavior through partial data traces represented as a subset of the data footprint of an application's execution. Partial data traces may be comparatively small and can be collected without prohibitively large overheads during execution, while complete data traces are expensive to generate and generally result in very large amounts of data.

This work focuses on the collection of partial address traces without compiler or linker support, *i.e.*, arbitrary executables can be subject to the generation of traces. We dynamically modify an executing application by injecting instrumentation code via binary rewriting. The instrumentation is placed at the point of memory accesses to precisely capture the data references issued by an application. In addition, the user may activate or deactivate tracing so that data reference streams are being generated or being suppressed, respectively. This facility builds the foundation for capturing partial memory traces. In the following, the software infrastructure for partial trace generation is detailed.

The METRIC framework is shown in Figure 1. The user provides the application process id (PID) and the names of the target function(s) to the control program. The controller attaches to the target and retrieves its Control Flow Graph (CFG). It parses the text section of the target for memory access instructions, *i.e.*, loads and stores. It uses the CFG to determine the scope structure of the target, *i.e.*, the function/loop entry and exit points and the nesting structure of loops. It then inserts instrumentation at memory access points and scope change instructions. 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 the instrumentation is complete, the target is allowed to continue. The handler functions in the shared library get invoked depending on the type of events occurring in the instrumented program, *i.e.*, load, store, enter_scope and exit_scope. The handler functions, in turn, call the compression routine which attempts to detect regular patterns in the incoming stream.

Once a specified number of events have been logged or a time threshold has been reached, the instrumentation is removed, and the target is allowed to continue. 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, and tags accesses to line num-

bers in the source. The cache simulator generates not only summary level information, but also reports detailed evictor information for source-related data structures, which is presented to the user, for analysis.

In our approach, we exploit source-related debugging information embedded in binaries for our analysis. 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 compilers and Intel/K&R's compiler 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, which are subject of our study, are not affected. Thus, compiling with symbolic information only increases executable size without significant degradation of performance.

3. Trace Generation and Compression

The generation of partial address traces provides the capability to later analyze this trace. We use a modified algorithm based upon our previous work [24] to obtain efficient runtime compression of this event trace. Our mechanism is tailored for regular data access patterns, such as those frequently occurring in tight loops. These patterns are represented via *regular section descriptors (RSDs)* as a tuple <*start_address, length, address_stride, event_type, start_sequence_id, sequence_id_stride,source_table_index*>, an extension of Havlak's and Kennedy's RSDs [13].

The `start_address`, `length` and `address_stride` describe the starting address, number of iterations and strides between successive address values generated by this pattern. The start position of the pattern in the overall event stream is indicated by the `start_sequence_id`, and its interleaving is described by the `sequence_id_stride`. The stride of RSDs may be an arbitrary function. We restrict ourselves to constants in this paper since we require fast online techniques to recognize RSDs. In different contexts, one may want to consider linear functions or higher order polynomials. Special access patterns are given by recurring references to a scalar or the same array element, which can be represented as RSDs with a constant stride of zero. The `event_type` distinguishes between reads, writes, enter_scope and exit_scope events. For the scope change events, the start_address field represents the scope id, and the address stride is zero. The `source_table_index` is an index into a table of (source_filename → line_number) tuples. It enables the cache simulator to correlate events with lines in the source code for user feedback.

Consider the example with a row-major layout shown in Figure 2. For the sake of simplicity, we assume an offset of one per array element. The read references to array `B` occur at offsets n+1, n+2, n+3 (corresponding to references `B[1,1], B[1,2] and B[1,3]`, respectively), for the first iteration of the outer loop and a length of n-1 accesses. The starting sequence id for the first access of the `B` array is 3 (since the first three events (seq_ids start from 0) are the two `enter_scopes` for scopes 1 and 2 as well as the read event for `A[i]`). For one iteration of the outer loop, accesses to the `B` array occur with an interleave distance of 3 in the overall event stream. Hence, the RSD for array `B` accesses for 1 iteration of the outer loop is:

`RSD5 <B+n+1,n-1,1,READ,3,3,3>`

Simple RSDs by themselves are not sufficiently expressive to capture the entire stream of accesses of either array A or B. To address this limitation, we extend this description by *power regular section descriptors (PRSDs)*, which allow the representation of power sets of RSDs as specified in Figure 2. A PRSD extends the tuple of an RSD, in that it may contain a PRSD (or RSD) itself, which represents the subset. The recursive structure of PRSDs provides a hierarchical means to represent recurring patterns with different start addresses but the same strides and lengths. This is useful for patterns that are usually encountered in nested loops.

The example in Figure 2 illustrates how all read accesses to array A can be combined as follows:

```
PRSD1:  <start_base_address = A,
         base_address_shift = 1,
         start_base_sequence_id = 2,
         base_sequence_id_shift = 3n-1,
         PRSD_length = n-1,RSD1>
```

This PRSD represents a total of n-1 repetitions of RSD1 with increments of 1 in addresses and interleaving distance of 3n-1 between the start of consecutive patterns in the overall event stream. Events, which cannot be classified as a part of a pattern, are represented by the *irregular access descriptors (IADs)* as: <*address, type, sequence_id, source_table_index*>. The `sequence_id` anchors the event in the overall event stream, and the `source_table_index` gives the (source_filename → line_number) mapping of the instruction causing this event. The `type` indicates event type (i.e enter / exit scope or load/ store). Line numbers are obtained from the binary's debug information, as explained earlier. Once a specified number of events have been logged or a time threshold has been reached, the instrumentation is removed, and the target is allowed to continue. The compressed description of the event trace (PRSDs & RSDs) is written to stable storage. Before we discusses the use of the compressed trace for cache simulation and user feedback, access ordering is detailed.

4. Ordering of Accesses

The previous section provided compact representations for regular access patterns within a sequence of data ref-

```
//Declare A[n], B[n][n], init. A w/ 0
  for (i = 0; i < n-1; i++)
  { // begin scope_1
    for(j = 0; j < n-1; j++)
    { // begin scope_2
        A[i] = A[i] + B[i+1][j+1];
    } // end scope_2
  } // end scope_1
```

Event Stream:

EnterScope1
  EnterScope2
    A[0] B[1][1] A[0]
    ...
  ExitScope2
  EnterScope2
    A[1] B[2][1] A[1]
    ...
  ExitScope2
ExitScope1

RSD:	<*start_addr, length, addr_stride,event_type, start_seq_id,seq_id_stride,source_table_index*>
PRSD:	<*base_addr, base_addr_shift, sequence_id_base, sequence_id_shift, PRSD_length,RSD*>

offsets in A:		Stream Representation:
reads:	000...	RSD1: <A, n-1,0,READ,2,3,1>
	111...	RSD2: <A+1, n-1,0,READ,3n+1,3,1>
		PRSD1: <A, 1,2,3n-1,n-1,RSD1>
writes:	000...	RSD3: <A, n-1, 0,WRITE,4,3,2>
	111...	RSD4: <A+1, n-1, 0,WRITE,3n+3,3,2>
		PRSD2: <A, 1, 4, 3n-1,n-1,RSD3>
offsets in B (reads only):		
n+1,n+2...		RSD5: <B+n+1, n-1, 1,READ,3,3,3>
2n+1 2n+2..		RSD6: <B+n+1,n-1, 1,READ,3n+2,3,3>
		PRSD3: <B+n+1, n-1, 3, 3n-1, n-1,RSD5>
For scope 2 : Enter_Scope ::		
1, 3n, 6n-1...		RSD7: <2,n-1,0,ENTER,1,3n-1,0>
For scope 2 : Exit_Scope ::		
3n-1, 6n-2...		RSD8: <2,n-1,0,EXIT,3n-1,3n-1,0>

Figure 2. Example: Representing Regular Access Patterns

erences. Data reference streams in numerical codes often exhibit accesses to multiple sequences in an interleaved manner. Consider the example in Figure 2 again: Accesses to elements of arrays `A` and `B` alternate (at different frequencies). We provide a compact, flexible representation that preserves the order of accesses through a PRSD, even across data structures. The `sequence_id` of RSDs specifies the order of the first occurrence for a reference. The `sequence_id_base` together with the `sequence_id_shift` determine the interleaving frequencies of different PRSDs. The former determine the base offset in the data reference stream while the latter specifies the next occurrence. To simplify the construction of a data stream from PRSDs, PRSDs are internally organized as a forest at the highest level, where each tree comprises a hierarchy of PRSDs with leaves representing RSDs.

The example in Figure 2 depicts the ordering of accesses by `sequence_id`s of 2, 4, 3, 0 and $3n-1$ for RSDs 1, 3, 5, 7 and 8, respectively. This corresponds to the original access order of entering the scope, repeatedly reading A and B as well as writing A before exiting the scope.

The abstraction of a data stream provides a compact representation of regular references within applications. Irregular accesses are represented separately in terms of an IAD, as explained before.

5. Online Detection of Access Patterns

This section describes our efficient online algorithm to detect RSDs [24]. This algorithm extracts the accesses corresponding to a data structure such as an array, despite the interleaving of alternate accesses to other data. In order to detect RSDs, a pool of references is maintained. The references lie within the *window* of addresses being scanned for potential RSDs. As new addresses are referenced, the window of active addresses advances within a pool. In order to determine RSDs with constant strides, it is imperative to compute differences between elements of the pool. To reduce the computational complexity, we store a *set of differences* along with each reference in the pool. The quest for locating RSDs reduces to one of finding a sequence of pool elements in which differences between addresses of stream elements are identical. The pool consisting of both the memory references and the calculated differences can be stored in a statically allocated, two-dimensional array, which is used in a circular manner by keeping track of two indices, the *start* and the *end* of the active addresses. The indices advance via modulo arithmetic through the pool. The pseudo code of the algorithm, omitting the details of aging and distinguishing access types, is presented in Figure 3.

The worst case complexity of the algorithm is $O(N \times w^2)$, where N is the number of references and w is the window size, which is a small constant. The innermost conditional results in constant time overhead due to hashing techniques. In practice, we observed linear dependence on w for benchmarks with regular accesses due to stream extensions. If a reference extends a (known) stream, then there is no need to compute differences for it, *i.e.*, by bookkeeping for the reservation pool, an $O(N)$ factor that is dominated by the $O(N \times w)$ stream table cost. Aging of streams can easily be achieved by including a tag with each tuple in the stream table signifying the stream's age.

We illustrate the application of the algorithm on the example in Figure 2. We assume `A` and `B` start at location `100`

```
WHILE new reference exists DO
  Increment column; /* move window */
  pool[0][column] := new reference; /* add ref. to pool */
  IF reference IN some RSD THEN
    Update length of RSD in stream table;
    Mark column in pool (shaded in example);
  ELSE /* compute and store differences in pool */
    FOR i := 1 TO w - 1 DO
      pool[i][column] := pool[0][column]-pool[0][column-i];
    END FOR;
    found := FALSE; /* find RSDs w/ min. length 3 */
    IF there exists i in 1..w-1 AND k in 1..w-1
      such that pool[i][column] == pool[k][column-i] THEN
      Enter RSD in stream table;
      Mark columns 0,i,k in pool (shaded in example);
    END IF;
  END IF;
END WHILE;
```

Figure 3. Online Algorithm to Detect RSDs

and `200`, respectively, and are stored in row-major layout. Let array elements occupy single memory locations. The accesses translate into the following address sequence distinguished by read(R) and write(W) accesses:
R100 R211 W100 ; R100 R212 W100 ; R100 R213 W100 ; ...
R101 R221 W101 ; R101 R222 W101 ; R101 R223 W101 ; ...
Figure 4 shows the snapshot of the pool as the first eight references are encountered. The header row shows the referenced locations. Each column contains the *difference* between the value in the current column header and the value in a preceding column (see "compute and store differences" in Figure 3). The particular element used for calculating the *difference* depends on the row in which the difference is computed and requires matching access types [24]. To capture RSDs within a window size `w`, we need only compute the differences above the diagonal of the pool table. On seeing the third `R100` (assuming a minimum length of three), we will identify an RSD by observing the two corresponding differences of `0` (circled) in a transitive relationship, resulting in RSD `<100, 3, 0, ...>` (shaded). Later `R100`s will extend this RSD in length. Similarly, a difference of `1` (circled) for references R211, R212 and R213 results in RSD `<211, 3, 1, ...>`. We only refer to the first three components of RSDs that contribute to the algorithm. Details of composing RSDs into PRSDs are also omitted since they are straight forward.

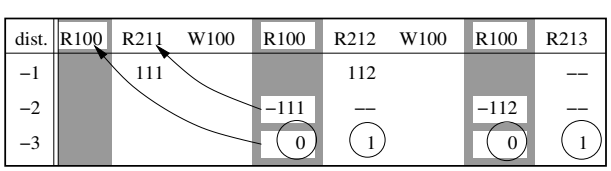


Figure 4. Snapshot of the Reservation Pool

6. Cache Simulation and User Feedback

The compressed event trace is used for off-line incremental cache simulation. We use a modified version of `MHSim` [22] as the cache simulator. MHSim was designed "to identify source program references causing poor cache utilization, quantify cache conflicts, temporal and spatial reuse, and correlate simulation results to references and loops in the source code".

The original MHSim package used a source-to-source Fortran translator to instrument data accesses with calls to the MHSim cache simulation routines. However, this strategy has several disadvantages. Data accesses specified in the source code are simulated in their canonical execution order ignoring any compiler transformations that may change the order of accesses. Additionally, the compiler may eliminate several accesses during optimizations (*e.g.*, common sub-expressions). We avoid these problems by instrumenting the application binary instead of the application source. The event trace describes the order of accesses as they occurred during execution. The cache simulator driver uses the application symbol table to reverse map the trace addresses to variable identifiers in the source. It relies on the symbolic information embedded in the binary, as explained before. Every compressed trace representation (*i.e.*, PRSDs, RSDs and IADs) has an associated "source_table_index", which indexes into a table of (source_filename $\rightarrow$ line_number) mappings correlating the access instruction in the binary to the source level access that it represents. MHSim is capable of simulating multiple levels of memory hierarchy. However, we concentrate our analysis only on the first level of cache (*i.e.*, L1 cache).

For each access point, MHSim provides:

- **total hits** associated with the reference.
- **total misses** associated with the reference.
- **miss ratio** for the reference: basic factor in evaluating locality of reference.
- **temporal reuse fraction** for the reference, *i.e.*, the number of $\frac{temporal\ hits}{total\ hits}$: Useful for determining how much locality (temporal and spatial) the reference is providing. This can be checked against the source code to see how much potential for locality the reference actually has.
- **spatial use**, which is computed as $\frac{used\ bytes}{block\ size} * number\ of\ evictions$, gives an indication of the fraction of the cache block being referenced before an eviction occurs. A low spatial use count would indicate that the machine is wasting cycles and/or space bringing in data that is never referenced.
- **evictor references**: the identities of the competing references, which evicted this reference from the cache, and their relative counts. Useful for determining which data objects conflict with each other. The conflict can

File	Line	Reference	Source_Ref	Hits	Misses	Miss Ratio	Temporal Ratio	Spatial Use
mm.c	63	**xz_Read_1**	**xz[k][j]**	**0**	**2.50e+05**	**1.00**	**no hits**	**0.171**
mm.c	63	**xy_Read_0**	**xy[i][k]**	2.39e+05	1.10e+04	0.0441	**0.854**	**0.129**
mm.c	63	**xx_Read_2**	**xx[i][j]**	2.50e+05	1.57e+02	0.000628	1.00	**0.5**
mm.c	63	**xx_Write_3**	**xx[i][j]**	2.50e+05	0.0	0.0	1.00	**no evicts**

Figure 5. Per-Reference Cache Statistics for Unoptimized Matrix Multiply

Reference				Evictors					
File	Line	Name	Source_Ref	File	Line	Name	Source_Ref	Count	Percent
mm.c	63	**xy_Read_0**	**xy[i][k]**	mm.c	63	**xz_Read_1**	**xz[k][j]**	**10863**	**100.00**
mm.c	63	**xz_Read_1**	**xz[k][j]**	mm.c	63	**xz_Read_1**	**xz[k][j]**	**238150**	**95.58**
				mm.c	63	**xy_Read_0**	**xy[i][k]**	10854	4.36
				mm.c	63	**xx_Read_2**	**xx[i][j]**	149	0.06
mm.c	63	**xx_Read_2**	**xx[i][j]**	mm.c	63	**xz_Read_1**	**xz[k][j]**	**149**	**100.00**
mm.c	63	**xx_Write_3**	**xx[i][j]**	mm.c	63	**xz_Read_1**	**xz[k][j]**	**149**	**100.00**

Figure 6. Evictor Information for Unoptimized Matrix Multiply

be resolved by program transformations or by data reorganization (*e.g.*, array padding).

7. Experiments

In the following section, we illustrate the use of our framework to analyze the locality behavior of several test kernels. We show how the cache simulation results can be used to detect and isolate problem areas and to derive appropriate program transformations.

The cache configuration for simulation was that of a MIPS R12000 processor with a total cache size of 32 KB, 32 byte line size and 2-way associativity. A partial data trace was obtained for each kernel. The compressed trace was run through the cache simulator to produce memory hierarchy statistics.

7.1. Matrix Multiplication (mm)

We first report on experiments with a matrix multiplication kernel. The C source code is shown below (assuming that arrays are row-major).

```
60 for (i=0; i < MAT_DIM; i++)
61   for (j = 0; j < MAT_DIM; j++)
62     for (k = 0; k < MAT_DIM; k++)
63       xx[i][j]=xy[i][k]*xz[k][j]+xx[i][j];
MAT_DIM = 800
total memory accesses logged = 1000000
```

The order of accesses is important to distinguish two different source code references to the same array in the report statistics (for example, `xx[i][j] READ` and `xx[i][j] WRITE`). In the report tables, each distinct reference point from the machine code is represented by an identifier composed of the name of the data object it refers to, appended with the type of access (READ/WRITE) and the position of the reference point in the overall order of accesses in the binary. (For example, in the untiled matrix multiply kernel's machine code, the order of accesses is xy(read), xz(read), xx(read), xx(write) indicated as `xy_Read_0`, `xz_Read_1`, `xx_Read_2` and `xx_Write_3`, respectively.)

We observe the following overall performance:

reads	=	750000	temporal hits	=	703930
writes	=	250000	spatial hits	=	34881
hits	=	738811	temporal ratio	=	0.95279
misses	=	261189	spatial ratio	=	0.04721
miss ratio	=	**0.26119**	**spatial use**	=	**0.16980**

The high miss rate (26%) should be the first indication of concern for the analyst. Interestingly, the spatial use value is quite low (0.16980). This indicates that the current program referencing order is inefficient in the sense that most cache blocks are being evicted before the entire data in the block is referenced at least once.

Let us explore the cache statistics at a higher level of detail. Figure 5 depicts the per-reference cache statistics. The `xz_Read_1` performance is immediately striking. All accesses to the xz array were misses. A look at the source indicates the cause: The k loop runs over the rows of xz. By the time reuse of xz data occurs (on next iteration of the i loop), the data has been flushed from the cache. With only a single element of the cache line containing xz being referenced for each iteration of k, the spatial use value is also low (0.171).

With the `xx_Read_2` reference, the number of hits is large, as expected, since the `xx[i][j]` read is invariant for the k loop. Even here, however, the spatial use is low (0.5) indicating premature eviction before all data in the block

File	Line	Reference	Source_Ref	Hits	Misses	Miss Ratio	Temporal Ratio	Spatial Use
mm.c	86	**xx_Read_2**	**xx[i][j]**	2.41e+05	8.79e+03	0.0352	0.972	**0.673**
mm.c	86	**xy_Read_0**	**xy[i][k]**	2.41e+05	8.79e+03	0.0352	0.896	**0.732**
mm.c	86	**xz_Read_1**	**xz[k][j]**	**2.50e+05**	**2.88e+02**	**0.0011**	**0.999**	**0.861**
mm.c	86	**xx_Write_3**	**xx[i][j]**	2.50e+05	0.00e+00	0.0	0.989	**no evicts**

Figure 7. Per-Reference Cache Statistics for Optimized Matrix Multiply

Reference				Evictors					
File	Line	Name	Source_Ref	File	Line	Name	Source_Ref	Count	Percent
mm.c	86	**xz_Read_1**	**xz[k][j]**	mm.c	86	xy_Read_0	xy[i][k]	100	69.44
				mm.c	86	xx_Read_2	xx[i][j]	42	29.17
				mm.c	86	**xz_Read_1**	**xz[k][j]**	**2**	1.39
mm.c	86	**xx_Read_2**	**xx[i][j]**	mm.c	86	xx_Read_2	xx[i][j]	4976	60.05
				mm.c	86	xy_Read_0	xy[i][k]	3297	39.79
				mm.c	86	**xz_Read_1**	**xz[k][j]**	**14**	0.17
mm.c	86	**xx_Write_3**	**xx[i][j]**	mm.c	86	xx_Read_2	xx[i][j]	4976	60.05
				mm.c	86	xy_Read_0	xy[i][k]	3297	39.79
				mm.c	86	**xz_Read_1**	**xz[k][j]**	**14**	0.17
mm.c	86	**xy_Read_0**	**xy[i][k]**	mm.c	86	xy_Read_0	xy[i][k]	5010	59.52
				mm.c	86	xx_Read_2	xx[i][j]	3279	38.96
				mm.c	86	**xz_Read_1**	**xz[k][j]**	**128**	1.52

Figure 8. Evictor Information for Optimized Matrix Multiply

was referenced. The `xx_Write_3` writes to data locations already brought into cache by the `xx_Read_2` reference, explaining a miss rate of 0.

For the `xy_Read_0` reference, the number of hits is quite large, comparable in magnitude to the hits for the `xx_Read_2` reference. A surprising feature is the relatively high temporal ratio (0.854). With the k loop running over the column dimension of xy and temporal reuse not occurring until the next iteration of j, we would instead expect the spatial fraction of hits to be high. This means that the `xy_Read_0` reference does not experience too much interference from other references over long stretches of accesses (more than the length of the k loop).

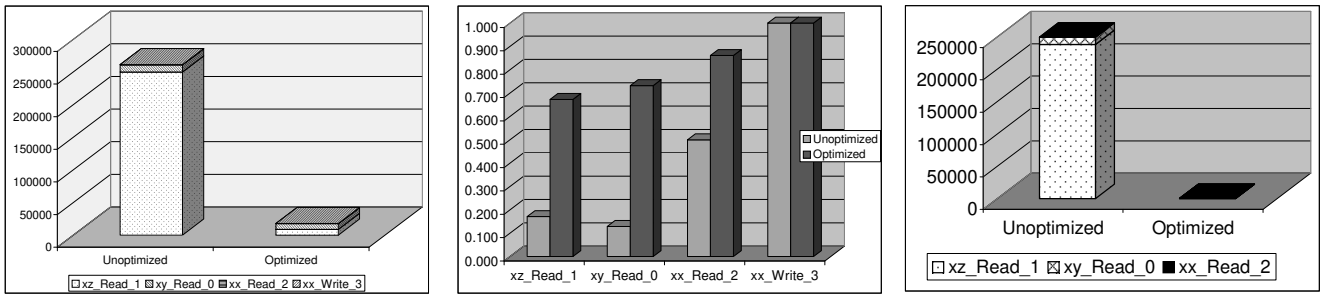
The evictor table for mm is shown in Figure 6. Again, the `xz_Read_1` reference performance is unusual. Over 95% of the time, `xz_Read_1` interfered with itself, indicating a capacity problem. Additionally, `xz_Read_1` was the evictor for all the other references (100% of the time). These evictions by xz cause premature invalidation of block data belonging to evicted references leading to low spatial use (and, thus, low overall cache usage) for these references.

Improving data locality: We have pinpointed the xz array references as having the maximum effect on cache performance. We need to change the program structure to reduce the access footprint for xz. By interchanging the `j` and `k` loops, we can increase locality for xz (since now the inner loop runs over the columns of xz), which has the highest number of misses. By strip mining the `j` and `k` loops, we can force the temporal reuse to occur at shorter intervals in the overall event stream, especially for arrays xy and xx. This will reduce the chance of these references having blocks flushed from the cache before the entire block data is utilized. The new transformed code with these improvements is shown below.

```
81 for (jj=0; jj<MAT_DIM; jj += ts)
82  for (kk=0; kk<MAT_DIM; kk += ts)
83   for (i=0; i<MAT_DIM; i++)
84    for (k=kk; k<min(kk+ts,MAT_DIM); k++)
85     for (j=jj; j<min(jj+ts,MAT_DIM); j++)
86      xx[i][j] = xy[i][k] * xz[k][j] +
                   xx[i][j];
tile size ts = 16;
```

We observe the following overall performance:

reads	=	750000	temporal hits	=	947173
writes	=	250000	spatial hits	=	34955
hits	=	982128	temporal ratio	=	0.96441
misses	=	17872	spatial ratio	=	0.03559
miss ratio	=	**0.01787**	**spatial use**	=	**0.70394**



(a) Total Number of Misses (b) Spatial Use per Reference (c) Evictors for xz_Read_1

Figure 9. Contrasted Metrics for Matrix Multiply before and after Optimizations

Figures 7 and 8 show the per-reference cache statistics and the evictor table for the transformed matrix multiply code. Figures 9(a-c) contrast the results before and after optimization for misses, use and evictor information for the critical reference `xz_Read_1`, respectively. The overall miss ratio has decreased two orders of magnitude from 0.26 to 0.017. The overall spatial use has also improved greatly from 0.16980 to 0.70394. The greatest improvement has occurred for the `xz_Read_1` reference; the number of hits has gone down from 0 to 2.5e+05, with 99.9% of these being temporal hits.

Also, for all references, the spatial use values have gone up, increasing the efficiency of cache usage. The eviction table in Figure 8 explains why this happened. The number of evictions for most references has gone down significantly, especially for the `xz` reference from almost 240,000 to less than 200. Evictors for this reference are also depicted in Figure 9(c). For other references, the evictors, in the majority of cases, are references to the same array. Overall, the interference between the xz reference and other references has been significantly reduced with a slight overall increase in interference between other references (*e.g.*, between `xy_Read_0` and `xx_Read_2`).

Consider the pseudo-code for the unoptimized matrix multiply again. Two references to `xx`, a read and a write, are executed on each array element. We performed our experiments by compiling without allocating `xx[i][j]` to a register in the inner loop. While register allocation would have affected the total number of references for `xx`, it has a negligible impact on eviction and miss ratios, as verified by the low eviction count of 149 in Figure 6. Only one out of 800 array references would have been affected in arrays `xy` and `xz`. In the optimized case, allocating `xy` to a register would have had a similar effect since the cache associativity was two and both tiled blocks of `xx` and `xy` could co-exist in cache.

7.2. Erlebacher ADI Integration

The C kernel for the Erlebacher ADI Integration is shown below. For this kernel, the optimizations possible (loop interchange and fusion) are visually apparent. However, we illustrate how the cache results can reveal the need for these optimizations. The result of the analysis would be similar in the case of more non-obvious codes benefiting from the same loop optimizations.

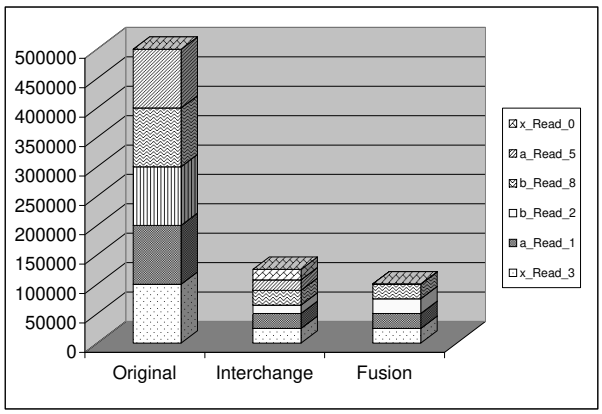
```
16 for (k = 1; k < N; k++) {
17   for (i = 2; i < N; i++)
18     x[i][k] = x[i][k] -
         x[i-1][k]*a[i][k] /b[i-1][k];
19   for (i = 2; i < N; i++)
20     b[i][k] = b[i][k] -
         a[i][k] * a[i][k] /b[i-1][k];
21 }
N = 800
total memory accesses logged = 1000000
```

We observe the following overall performance:

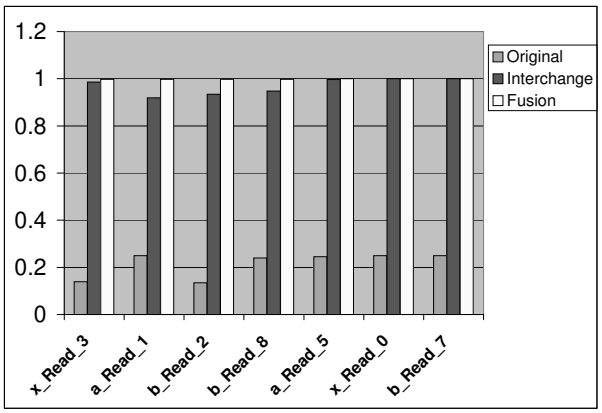
reads	=	800000	temporal hits	=	351731
writes	=	200000	spatial hits	=	147768
hits	=	499499	temporal ratio	=	0.70417
misses	=	500501	**spatial ratio**	=	**0.29583**
miss ratio	=	**0.50050**	**spatial use**	=	**0.20181**

As in `mm`, the primary indicator of concern is the miss ratio — over 50% of the total accesses are misses. Spatial hits constitute just a third of the overall hits. The low spatial use value (0.20) indicates the poor efficiency of the current program order of memory accesses.

The reference-specific statistics are summarized in the first bar of Figure 10(a). In addition, Figure 10(b) indicates low spatial use for read references in the original code. The first five references `x[i][k]`, `a[i][k]`, `b[i-1][k]`,`b[i][k]` and `a[i][k]` do not have a single hit in the cache. Looking at the source code, a common



(a) Total Number of Misses



(b)Spatial Use per Reference

Figure 10. Contrasted Metrics for ADI before and after Optimizations

pattern is evident among all these reference: the inner loop (i loop) runs over the rows of these references. Spatially adjacent elements from these arrays, in the same cache block as these references, are accessed only on the next iteration of the k loop, by which time they have been flushed from the cache. Hence, the spatial use value is low, and spatial hits are negligible.

The evictor information (not shown due to its size) actually indicates this problem independent of source code knowledge. A circular dependency exists for the references and their evictors within both inner loops. We need to reorder the accesses so that we can take advantage of spatial reuse by running the inner loop over the columns (rather than rows) of these references. The source code indicates that this is possible without violating data dependencies.

Improving Locality: The loop-interchanged kernel is shown below.

```
16 for (i = 2; i < N; i++)
17   for (k = 1; k < N; k++)
18     x[i][k] = x[i][k] -
         x[i-1][k] * a[i][k] /b[i-1][k];
19   for (k = 1; k < N; k++)
20     b[i][k] = b[i][k] -
         a[i][k] * a[i][k] /b[i-1][k];
21 }
```

We observe the following overall performance:

reads	=	800000	temporal hits	=	454867
writes	=	200000	spatial hits	=	419733
hits	=	874600	temporal ratio	=	0.52009
misses	=	125400	spatial ratio	=	0.47991
miss ratio	=	**0.12540**	**spatial use**	=	**0.96281**

There is significant improvement in the miss ratio: it has fallen from 50% to less than 13% in the optimized code. The access efficiency, indicated by the spatial use, has increased drastically from 0.20 to 0.96.

Can we optimize the locality further? To determine this, we need to look at the reference-specific statistics, summarized for selected references in the second bar of Figure 10(a). The miss ratio has decreased substantially, especially for the five references we focused on (`x_Read_3, a_Read_1, b_Read_2, b_Read_8, a_Read_5`) in the analysis of the unoptimized kernel. However, there still remain a non-negligible number of misses. If we look at the source names for the references, we see that there are a lot of common expressions (especially `a[i][k]` and `b[i][k]`). Grouping these accesses together would further increase locality for the secondary accesses to the same array (*e.g.*, grouping `a_Read_1` and `a_Read_5` would eliminate misses for `a_Read_5`). Of course, this transformation would be possible only if no data dependencies are violated. The new kernel is shown below.

```
14 for (i = 2; i < N; i++)
15   for (k = 1; k < N; k++) {
16     x[i][k] = x[i][k] -
         x[i-1][k] * a[i][k] / b[i-1][k];
17     b[i][k] = b[i][k] -
         a[i][k] * a[i][k] / b[i-1][k];
18   }
```

We observe the following overall performance:

reads	=	800000	temporal hits	=	549822
writes	=	200000	spatial hits	=	349849
hits	=	899671	temporal ratio	=	0.61114
misses	=	100329	spatial ratio	=	0.38886
miss ratio	=	**0.10033**	**spatial use**	=	**0.99798**

The miss ratio has decreased from 12.5% to 10%. The temporal use increased due to grouping of accesses, lead-

ing to approximately 5% increase in temporal hits. As a side-effect of the reduced number of evictions (directly correlated to reduction in total misses), the spatial use has increased to `0.997`, indicating excellent access efficiency.

The last bar in Figure 10(a) shows the per-reference statistics for the loop-fused case. The table indicates that the chief improvement has been in the `a_Read_5` and `x_Read_0` references. Grouping the `a[i][k]` access for `a_Read_5` and `a_Read_1` caused the misses for `a_Read_5` to go down to zero. The `x_Read_0` reference also decreased its number of misses by over two orders of magnitude, leading to a miss ratio of almost 0. This is surprising since the reuse for the `x[i-1][k]` element (due to the `x[i][k]` read reference) occurs only on the next iteration of the i loop. The reduction in the overall misses (and, thus, the evictions) due to grouping seems to have reduced the cross-interference for the `x[i-1][k]` reference as a side effect.

Careful analysis of the statistics reveals there is still potential for improvement. The `x_Read_3` (`x[i][k]`) and `x_Read_0` (`x[i-1][k]`) as well as `b_Read_2` (`b[i-1][k]`) and `b_Read_8` (`b[i][k]`) share temporal reuse potential on adjacent iterations of the i loop. The misses for `x_Read_0` and `b_Read_8` can be reduced by tiling (blocking) for the i and k loops. However, we will not discuss these modifications here.

8. Related Work

The idea of enhancing DynInst by supplying the register contents of scratch and non-scratch registers and the ability to invoke high-level routines through indirect calls to dynamically loaded shared libraries builds on our prior work on multi-threaded debugging [26]. The performance improvements due to inline instrumentation are consistent with previously published techniques for supporting fast breakpoints [16]. DynInst uses techniques similar to fast breakpoints for inline instrumentation but, in contrast to the original work on fast breakpoints, in a portable fashion. The invocation of arbitrary routines has also been realized in a similar fashion in DPCL, a distributed instrumentation framework on top of DynInst [10].

Regular Section Descriptors represent a particular instance of a common concept in memory optimizations, either in software or hardware. For instance, RSDs [13] are virtually identical to the *stream descriptors* used at about the same time in the compiler and memory systems work inspired by the WM architecture [34].

Atom has been widely used as a binary rewriting tool to statically insert instrumentation code into application binaries [28]. Dynamic binary rewriting enhances this approach by its ability to dynamically select place and time for instrumentations. 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.* [31] 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 [3]. 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 search (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 IA-64), 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. Finally, this described hardware support is not portable. In contrast, our approach to generating traces is applicable to today's 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 are presently not supported in contemporary architectures [15, 23].

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 [22]. This approach shares its goal of cache correlation with our work, and we are considering collaborative efforts. CProf [19] is a similar tool that relies on post link-time binary editing through EEL [17, 18] 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 [20]. Our work differs in the fundamental approach of rewriting binaries, which

is neither restricted to a special compiler or programming language, nor does it preclude the analysis of library routines. Another major difference addresses the overhead of large data traces inherent to all these approaches. We restrict ourselves to partial traces and employ trace compression to provide compact representations.

Recent work by Chilimbi *et al.* concentrates on language support and data layout to better exploit caches [7, 6] as well as quantitative metrics to assess memory bottlenecks within the data reference stream [5]. This work introduces the term *whole program stream* (WPS) to refer to the data reference stream, and presents methods to compactly represent the WPS in a grammatical form. However, the WPS compression is only applicable to scalar data, while our approach addresses compact representations for array accesses and even dynamically allocated objects. Other efforts concentrate on access modeling based on whole program traces [2, 14] using cache miss equations [11] or symbolic reference analysis at the source level based on Presburger formulas [4]. These approaches involve linear solvers with response times on the order of several minutes up to over an hour. We concentrate our efforts on providing feedback to a programmer quickly.

A number of approaches address dynamic optimizations through binary translation and just-in-time compilation techniques for native code [27, 1, 8, 29, 12]. The main thrust of these techniques is program transformation based on knowledge about taken execution paths, such as trace scheduling. The transformations include the reallocation of registers and loop transformations (such as code motion and unrolling), to name a few. These efforts are constrained by the trade-off 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 [9]. 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 metric 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). Their approach differs in that they neither capture partial data traces nor present a concept for such an approach. Their compression algorithm is inferior since it results in linear space representations for interleaved patterns, such as matrices sequentially indexed, whereas constant space suffices, as demonstrated by our algorithm and Figure 2. Our cache analysis is more powerful. It reports not only per-reference metric but also per-reference evictor information, which is imperative to infer potential for optimizations. Subsequently, we are able to apply more sophisticated optimizations, such as tiling and loop transformations.

9. Future Work

METRIC represents the first step towards a tool that alters long-running programs on-the-fly so that their speed increases over its execution time – without any recompilation or user interaction. We are currently working on the second step, the applications of program analysis and subsequent dynamic optimizations on the binary. As such, automated optimization and on-the-fly injection of optimized code present work in progress. The former requires not only the reconstruction of the control-flow graph, which is already available at the binary level, but also the calculation of data-flow information and the detection of induction variables in order to infer data dependencies and dependence distance vectors [32, 33], a prerequisite to determine if certain program transformations preserve the semantics.

10. Conclusion

In this paper, 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, the generation and compression of partial data traces as well as the simulation of reference behavior in terms of caching. By correlating evictor information and aggregated cache metrics, sources of inefficiencies can be localized. The analysis allows us to infer the potential for program transformations. These transformations result in an absolute miss rate reduction of up to 40%. Our results still use manual code transformations but we are working on an automated approach to optimize applications on-the-fly, a task that faces many interesting challenges.

Acknowledgments

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

References

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

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

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

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

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

[6] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, May 1999.

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

[8] C. Cifuentes and M. V. Emmerik. UQBT: Adaptable binary translation at low cost. *Computer*, 33(3):60–66, Mar. 2000.

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

[10] L. DeRose, J. Hollingsworth, and T. Hoover. The dynamic probe class library – an infrastructure for developing instrumentation for performance tools. In *International Parallel and Distributed Processing Symposium*, Apr. 2001.

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

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

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

[14] J. Hollingsworth and L. DeRose. The sigma tools. In *Paradyn/Condor Week*, Mar. 2001.

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

[16] P. B. Kessler. Fast breakpoints: Design and implementation. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 78–84, 1990.

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

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

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

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

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

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

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

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

[25] I. Research. Personal communications. July 2002.

[26] D. Schulz and F. Mueller. A thread-aware debugger with an open interface. In *ACM International Symposium on Software Testing and Analysis*, pages 201–211, Sept. 2000.

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

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

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

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

[31] D. Weikle, S. McKee, K. Skadron, and W. Wulf. Caches as filters: A framework for the analysis of caching systems. In *Grace Murray Hopper Conference*, Sept. 2000.

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

[33] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[34] W. Wulf. Evaluation of the wm architecture. In *International Symposium on Computer Architecture*, pages 382–390, May 1992.