Detecting Memory Performance Bottlenecks via Binary Rewriting

Jaydeep Marathe and Frank Mueller

Department of Computer Science
North Carolina State University
448 EGRC, Raleigh, NC 27695-7534

email: jpmarath@unity.ncsu.edu, mueller@cs.ncsu.edu

Abstract—

Binary rewriting has enjoyed increasing popularity for on-the-fly optimization/JITing as well as translation across different ISAs [1], [2], [3], [4], [5]. We examine a novel application of binary rewriting, namely, we deploy binary manipulation techniques to pin-point performance problems in the memory hierarchy. We extract reference information from applications in hot spots of execution and analyze the cache behavior with respect to the cause of misses. By correlating miss information to data structures, we can infer potential for improvement. Consequently, program transformations can be applied that alleviate prior performance problems in the memory hierarchy. In this context, dynamic binary rewriting techniques are utilized to gather partial data traces. Overall, our work is aimed at better exploiting existing memory bandwidth, thereby addressing increasing problems posed by the "memory wall", *i.e.*, by the increasing gap between processor and memory speeds.

I. 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 are experiencing as much as 50% stall cycles for repetitive data-centric tasks, in case of server workloads even up to 70% stalls [6]. The objective of this work is to determine the sources of inefficiencies in utilizing memory hierarchies and, consequently, to optimize memory behavior. The methodology to perform this analysis relies on our preliminary results on binary rewriting [7]. It is also influenced by our experience with large-scale benchmarks [8], which are another example for data-centric computations where the size of data exceeds cache capacities.

Binary manipulation techniques offer new opportunities to address this problem. They support program transformations tailored toward certain program inputs, and these transformations have been shown to potentially yield performance gains beyond the scope of static code optimization without profile-guided feedback [9]. In addition, transformations can be performed on the entire executable, including library routines and mixed-language application, such as commonly found in scientific production codes [8]. We focus on program analysis through instrumentation, another application of binary manipulation, and program transformation via *dynamic binary rewriting*.

Our work builds on the instrumentation framework DynInst [10]. We have extended this framework to instrument arbitrary instructions while preserving the instrumentation context [7]. In addition, inline instrumentation routines allow the customization of data collection as part of the application's executions, *i.e.*, without incurring context switch overhead as in the traditional mutator/controller framework of DynInst. This allows us to selectively instrument load/store instructions, generate compressed address traces online and process these traces offline. We also contribute a cache analysis approach, based on prior work [11], 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. In the following, we describe our preliminary experience in extracting partial data traces from applications and performance analysis to detect memory bottlenecks. We also show how the potential for improvement can be inferred from this data. Subsequently, program transformations can be performed, which yield significantly higher cache performance. In this work, we report our preliminary experience with an automated approach to generate partial data traces by means of dynamic binary rewriting. The program transformation step, currently subject to manual interaction, is currently being automated.

The paper is structured as follows. First, our binary instrumentation framework is introduced. Next, the generation, representation and compression of partial data traces is summarized. Then, incremental cache simulation for these partial traces is performed, and metrics to assess memory throughput are discussed. Experiments for two kernels demonstrate the potential to detect memory bottlenecks by analyzing the information of the cache simulation, and improvements for these kernels are reported after the codes are optimized with regard to their memory inefficiencies. Finally, we discuss future research directions and summarize our findings.

II. Target Program Instrumentation

Our instrumentation tool is based on *DynInst* [10], a component middleware designed primarily for "debugging, performance monitoring and application composition out of existing packages" for target instrumentation. DynInst provides a platform independent framework for dynamically modifying the running target application through bi-

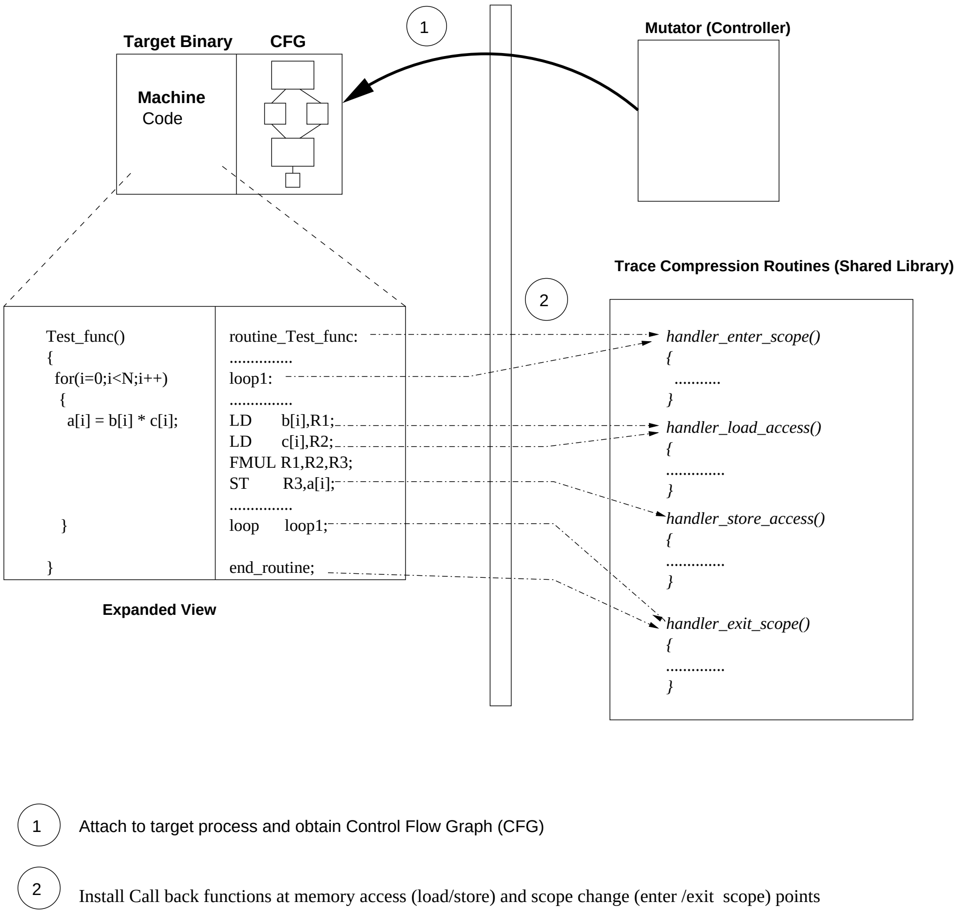

Fig. 1. Target Binary Instrumentation

nary rewriting.

At invocation time of our tool, the user provides the target application process id (PID) and the name(s) of the target function(s) to the control program. The control program uses DynInst to attach to the target and insert calls to callback functions at points of interest. The call back functions are in a shared library object. A one-shot instrumentation snippet is inserted to load the shared object into the application's address space.

The Control Flow Graph (CFG) of a hot spot function in the application is parsed to determine its scope structure. A *scope* is a distinct subdivision used by the cache simulator to report cache statistics. *Loop scopes* encompass natural loops while *Routine scopes* encompass entire functions. Entry and exit points of scopes are seeded with calls to handler routines in the shared library. In addition, the nesting of the scopes relative to each other is determined. Each scope (except the outermost scope) has a parent scope, and many scopes can have the same parent. The scope nesting enables the cache simulator to aggregate cache statistics. In addition, the raw machine code of the function is parsed to determine memory access instructions, *i.e.*, loads and stores. These points are also seeded with calls to handler routines in the shared library, as depicted in Figure 1.

After application instrumentation is complete, the target is allowed to continue. The event handlers 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.

In the following, 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.

III. Trace Generation and Compression

We use a modified algorithm based upon our previous work [7] to obtain efficient runtime compression of this event trace. Our mechanism is tailored for regular data access patterns, such as those frequently occurring

```
//Declare A[n], B[n][n], initialize A with 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:
enter_scope_1
enter_scope_2 A[0] B[1][1] A[0] ... exit_scope_2
enter_scope_2 A[1] B[2][1] A[1] ... exit_scope_2
...
exit_scope_1

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>

Fig. 2. Example: Representing Regular Access Patterns

in tight loops. These patterns are represented via *regular section descriptors (RSDs)* [12] as a tuple <*start_address, length, address_stride, event_type, start_sequence_id, sequence_id_stride,source_table_index*>.

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 `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 $\rightarrow$ 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. *Power regular section descriptors (PRSDs)* allow the representation of power sets of RSDs, as shown in Figure 2. A PRSD is useful for defining recurring patterns with different start addresses but same stride and lengths, which usually occur in nested loops.

The example in Figure 2 illustrates how all read accesses to array A can be combined in the following PRSD.

```
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 $\rightarrow$ line_number) mapping of the instruction causing this event.[1] The `type` indicates event type (i.e enter / exit scope or load/ store).

Once a specified number of events have been logged, the instrumentation is removed, and the target is allowed to continue. The compressed description of the event trace (*i.e.*, the PRSDs & RSDs) is written to stable storage.

The next section discusses the use of the compressed trace for cache simulation and user feedback.

[1] Line numbers are obtained from debugging information in the binary, as explained earlier.

IV. Cache Simulation and User Feedback

The compressed event trace is used for off-line incremental cache simulation. We use a modified version of `MHSim` [11] 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 → 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 be resolved by program transformations or by data reorganization (*e.g.*, array padding).

V. 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.

A. *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).

```
for (i=0; i < MAT_DIM; i++)
{
  for (j = 0; j < MAT_DIM; j++)
  {

    for (k = 0; k < MAT_DIM; k++)
    {
      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
writes     =   250000
hits       =   738811
misses     =   261189
```

File	Line	Reference	Source_Ref	Hits	Misses	Miss Ratio	Temporal Ratio	Spatial Use
mm.c	67	**xz_Read_1**	**xz[k][j]**	**0**	**2.50e+05**	**1.00**	**no hits**	**0.171**
mm.c	67	**xy_Read_0**	**xy[i][k]**	2.39e+05	1.10e+04	0.0441	**0.854**	**0.129**
mm.c	67	**xx_Read_2**	**xx[i][j]**	2.50e+05	1.57e+02	0.000628	1.00	**0.5**
mm.c	67	**xx_Write_3**	**xx[i][j]**	2.50e+05	0.0	0.0	1.00	**no evicts**

Fig. 3. Per Reference Cache Statistics for Unoptimized Matrix Multiply

	Reference				Evictors				
File	Line	Name	Source_Ref	File	Line	Name	Source_Ref	Count	Percent
mm.c	67	**xy_Read_0**	**xy[i][k]**	mm.c	67	**xz_Read_1**	**xz[k][j]**	**10863**	**100.00**
mm.c	67	**xz_Read_1**	**xz[k][j]**	mm.c	67	**xz_Read_1**	**xz[k][j]**	**238150**	**95.58**
				mm.c	67	**xy_Read_0**	**xy[i][k]**	10854	4.36
				mm.c	67	**xx_Read_2**	**xx[i][j]**	149	0.06
mm.c	67	**xx_Read_2**	**xx[i][j]**	mm.c	67	**xz_Read_1**	**xz[k][j]**	**149**	**100.00**
mm.c	67	**xx_Write_3**	**xx[i][j]**	mm.c	67	**xz_Read_1**	**xz[k][j]**	**149**	**100.00**

Fig. 4. Evictor Information for Unoptimized Matrix Multiply

miss ratio = 0.26119

```
temporal hits    =   703930
spatial hits     =    34881
temporal ratio   =  0.95279
spatial ratio    =  0.04721
```

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. Refer to Figure 3 for 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 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 till next iteration of j, we would rather 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 4. 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 efficiency) 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.

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

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**

Fig. 5. 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

Fig. 6. Evictor Information for Optimized Matrix Multiply

We observe the following overall performance:

```
reads                = 750000
writes               = 250000
hits                 = 982128
misses               = 17872
```

miss ratio = 0.01787

```
temporal hits        = 947173
spatial hits         = 34955
temporal ratio       = 0.96441
spatial ratio        = 0.03559
```

spatial use = 0.70394

Figures 5 and 6 show the per reference cache statistics and the evictor table for the transformed matrix multiply code. 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 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 below shows why this happened. The number of evictions for most references has gone down significantly, especially for the `xz` reference from more than 240,000 to less than 200. 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 inflicted on one 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 4. 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 affect since the cache associativity was two and both tiled blocks of `xx` and `xy` could co-exist

File	Line	Reference	Source_Ref	Hits	Misses	Miss Ratio	Temporal Ratio	Spatial Use
adi.c	18	**x_Read_3**	**x[i][k]**	**0.00e+00**	**1.00e+05**	**1.00**	**no hits**	**0.139**
adi.c	18	**a_Read_1**	**a[i][k]**	**0.00e+00**	**1.00e+05**	**1.00**	**no hits**	**0.250**
adi.c	18	**b_Read_2**	**b[i-1][k]**	**0.00e+00**	**1.00e+05**	**1.00**	**no hits**	**0.135**
adi.c	23	**b_Read_8**	**b[i][k]**	**0.00e+00**	**9.98e+04**	**1.00**	**no hits**	**0.240**
adi.c	23	**a_Read_5**	**a[i][k]**	**0.00e+00**	**9.98e+04**	**1.00**	**no hits**	**0.245**
adi.c	18	x_Read_0	x[i-1][k]	1.00e+05	1.26e+02	0.001	1.0	**0.250**
adi.c	23	b_Read_7	b[i-1][k]	9.96e+04	1.25e+02	0.001	1.0	**0.250**
adi.c	18	x_Write_4	x[i][k]	1.00e+05	0.00e+00	0.0	0.502	no evicts
adi.c	23	b_Write_9	b[i][k]	9.98e+04	0.00e+00	0.0	0.273	no evicts
adi.c	23	a_Read_6	a[i][k]	9.98e+04	0.00e+00	0.0	0.746	no evicts

Fig. 7. Per Reference Cache Statistics for Unoptimized ADI Integration

in cache.

B. 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 be used to divine the need for these optimizations. The result analysis would be similar in the case of more non-obvious codes benefiting from the same loop optimizations.

```
14 for (k = 1; k < N; k++)
15 {
16   for (i = 2; i < N; i++)
17   {
18     x[i][k] = x[i][k] -
         x[i-1][k]*a[i][k] /b[i-1][k];
19   }
20
21   for (i = 2; i < N; i++)
22   {
23     b[i][k] = b[i][k] -
         a[i][k] * a[i][k] /b[i-1][k];
24   }
25 }
N = 800
total memory accesses logged = 1000000
```

We observe the following overall performance:

```
reads               = 800000
writes              = 200000
hits                = 499499
misses              = 500501
```

miss ratio = 0.50050

```
temporal hits       = 351731
spatial hits        = 147768
temporal ratio      = 0.70417
```

spatial ratio = 0.29583
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 shown in Figure 7. 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 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, spatial use value is low, and spatial hits are negligible.

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.

```
14 for (i = 2; i < N; i++)
15 {
16   for (k = 1; k < N; k++)
17   {
18     x[i][k] = x[i][k] -
         x[i-1][k] * a[i][k] /b[i-1][k];
19   }
20
21   for (k = 1; k < N; k++)
22   {
23     b[i][k] = b[i][k] -
         a[i][k] * a[i][k] /b[i-1][k];
24   }
25 }
```

File	Line	Reference	Source_Ref	Hits	Misses	Miss Ratio	Temporal Ratio	Spatial Use
adi.c	18	**x_Read_3**	**x[i][k]**	**7.51e+04**	**2.51e+04**	**0.250**	**0.0**	**0.986**
adi.c	18	**a_Read_1**	**a[i][k]**	**7.51e+04**	**2.51e+04**	**0.250**	**0.0**	**0.919**
adi.c	23	**b_Read_8**	**b[i][k]**	**7.48e+04**	**2.50e+04**	**0.251**	**0.0**	**0.948**
adi.c	18	x_Read_0	x[i-1][k]	8.23e+04	1.79e+04	0.179	0.350	**1.00**
adi.c	23	**a_Read_5**	**a[i][k]**	**8.20e+04**	**1.78e+04**	**0.178**	**0.305**	**0.997**
adi.c	18	**b_Read_2**	**b[i-1][k]**	**8.57e+04**	**1.45e+04**	**0.145**	**0.495**	**0.934**
adi.c	23	b_Read_7	b[i-1][k]	9.98e+04	0.00e+00	0.0	0.927	no evicts
adi.c	18	x_Write_4	x[i][k]	1.00e+05	0.00e+00	0.0	0.876	no evicts
adi.c	23	a_Read_6	a[i][k]	9.98e+04	0.00e+00	0.0	0.911	no evicts
adi.c	23	b_Write_9	b[i][k]	9.98e+04	0.00e+00	0.0	0.876	no evicts

Fig. 8. Per Reference Cache Statistics for Loop-interchanged ADI Integration

We observe the following overall performance:

```
reads               = 800000
writes              = 200000
hits                = 874600
misses              = 125400
```

miss ratio = 0.12540

```
temporal hits       = 454867
spatial hits        = 419733
temporal ratio      = 0.52009
spatial ratio       = 0.47991
```

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, shown in Figure 8. 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, they still have 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.

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

We observe the following overall performance:

```
reads               = 800000
writes              = 200000
hits                = 899671
misses              = 100329
```

miss ratio = 0.10033

```
temporal hits       = 549822
spatial hits        = 349849
temporal ratio      = 0.61114
spatial ratio       = 0.38886
```

spatial use = 0.99798

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

Figure 9 shows the per reference statistics. 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 a slight scope 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.

File	Line	Reference	Source_Ref	Hits	Misses	Miss Ratio	Temporal Ratio	Spatial Use
adi.c	16	**x_Read_3**	**x[i][k]**	7.49e+04	2.51e+04	0.251	0.0	0.998
adi.c	16	**a_Read_1**	**a[i][k]**	7.49e+04	2.51e+04	0.251	0.0	0.998
adi.c	16	**b_Read_2**	**b[i-1][k]**	7.49e+04	2.51e+04	0.251	0.0	0.998
adi.c	17	**b_Read_8**	**b[i][k]**	7.49e+04	2.51e+04	0.251	0.0	0.998
adi.c	16	**x_Read_0**	**x[i-1][k]**	**9.99e+04**	**1.25e+02**	**0.001**	0.996	no evicts
adi.c	17	**a_Read_5**	**a[i][k]**	**1.00e+05**	**0.00e+00**	**0.0**	0.876	no evicts
adi.c	17	a_Read_6	a[i][k]	1.00e+05	0.00e+00	0.0	1.00	no evicts
adi.c	17	b_Write_9	b[i][k]	1.00e+05	0.00e+00	0.0	0.877	no evicts
adi.c	17	b_Read_7	b[i-1][k]	1.00e+05	0.00e+00	0.0	0.876	no evicts
adi.c	16	x_Write_4	x[i][k]	1.00e+05	0.00e+00	0.0	0.874	no evicts

Fig. 9. Per Reference Cache Statistics for Loop-fused ADI Integration

VI. Future Work

The reverse-mapping from addresses to variables works currently for local variables and for global variables, only. We are looking at ways to extend the framework to handle dynamically allocated variables as well. Partial data traces have unique challenges when dealing with dynamically allocated objects. Since we attach to the target dynamically, the standard approach of instrumenting the memory allocation routine (malloc) is not guaranteed to work since the target program counter may already have passed the object allocation stage. Also, the partial trace may be initiated from within the scope structure. Hence, there might be several "dangling" scopes, which were not completely entered or exited. This problem can be dealt with by discarding initial RSDs until a "stable" pattern is detected. The MHSim driver needs a graceful way to recover from this, *e.g.*, by a prior filter that consumes theses initial stray references. We also aim at automating the reverse-engineering of the memory access site to the source code reference that it represents in order to enhanced user feedback. Finally, automated optimization and on-the-fly injection of optimized code presents 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 [13], [14], which are a prerequisite to determine if certain program transformations preserve the semantics.

VII. Conclusion

We have showed 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 in hot spots of execution, 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, which can be applied on the binary and injected into the running application. While our preliminary results still use manual code transformations in response to this automated analysis, we are working on an automated approach to improve memory performance of applications on-the-fly, a task that faces many interesting challenges.

References

[1] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, "Binary translation," *Communications of the ACM*, vol. 36, no. 2, pp. 69–81, Feb. 1993.

[2] Cristina Cifuentes and Mike Van Emmerik, "UQBT: Adaptable binary translation at low cost," *Computer*, vol. 33, no. 3, pp. 60–66, Mar. 2000.

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

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

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

[6] Intel Research, "Personal communications," July 2002.

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

[8] 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.

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

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

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

[12] Paul Havlak and Ken Kennedy, "An implementation of interprocedural bounded regular section analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 350–360, July 1991.

[13] M. Wolf and M. Lam, "A data locality optimizating algorithm," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1991, pp. 30–44.

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