Partial Data Traces: Efficient Generation and Representation *

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

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

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

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

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

Abstract

Binary manipulation techniques are increasing in popularity. They support program transformations tailored toward certain program inputs, and these transformations have been shown to yield performance gains beyond the scope of static code optimizations without profile-directed feedback. They even deliver moderate gains in the presence of profile-guided optimizations. In addition, transformations can be performed on the entire executable, including library routines. This work focuses on program instrumentation, yet another application of binary manipulation.

This paper reports preliminary results on generating partial data traces through *dynamic* binary rewriting. The contributions are threefold. First, a portable method for extracting precise data traces for partial executions of arbitrary applications is developed. Second, a set of hierarchical structures for compactly representing these accesses is developed. Third, an efficient online algorithm to detect regular accesses is introduced. We utilize dynamic binary rewriting to collect partial address traces of regions within a program selectively. This allows partial tracing of hot paths for only a short time during program execution in contrast to static rewriting techniques that lack hot path detection and also lack facilities to limit the duration of data collection. Preliminary results show considerable reductions in performance overhead over a course of experiments as well as the ability to represent regular access patters of nested loops as hierarchical structures of constant size. These efforts are part of a larger project to counter the increasing gap between processor and main memory speeds by means of software optimization and hardware enhancements.

*Part of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48, UCRL-JC-144405.

1. Introduction

The manipulation of the binary representation of executable programs is becoming increasingly important. Dynamic compilation techniques, such as just-in-time compilation, are but one example of binary translation. Examples include Jalapeño [1] at the level of virtual machines and Dynamo [2] for native code. Quantitative results reported for Dynamo underline the effectiveness of dynamic compilation, even for native code, since it has been shown to outperform profile-directed feedback compilation techniques. In addition, profile-directed feedback requires multiple compilations, which has found only limited acceptance among users. Dynamic compilation can be performed on the binary representation during program execution, and does not require recompilation.

A new trend in the manipulation of binary executables is the area of dynamic instrumentation, which shares aspects of its motivation as well as methods of implementation with dynamic compilation. Traditional instrumentation generally requires compiler interaction (*e.g.*, for profiling) or the inclusion of special libraries (*e.g.*, for heap monitoring). Dynamic instrumentation removes the requirements of recompiling or relinking. The techniques for dynamic instrumentation are based on modifications of an application during execution. For example, our work builds on an instrumentation framework, DynInst [3], that relies on techniques of *dynamic binary rewriting* during program execution.

Binary rewriting is a term that generally refers to *post-link-time* modifications of an executable, *i.e.*, the application's binary representation, before running the program [25, 17, 18]. In contrast, binary translation represents the process of modifying the instructions (and, although less frequently exercised, also the data) of an application *while it is executing* [24, 9]. Binary translation typically involves the translation from the instruction set of one architecture to another architecture as execution paths are being touched. Similar techniques are used in dynamic code optimization to discover hot paths [2]. *Dynamic binary rewriting* is a combination of these approaches that applies to the efforts of our work. Dynamic binary rewriting uses a control process to *rewrite* the binary representation of an *executing* application process. However, during the rewrite process the execution of the application is briefly suspended before it resumes executing where it was interrupted. In contrast, binary translation and dynamic code optimization modify the application from *within* the application, *e.g.*, just-in-time compilation is part of the application's execution. Finally, traditional (static) binary rewriting modifies a binary representation *before* execution.

We employ dynamic binary rewriting techniques for extracting footprints of data references from an application's execution. This work is motivated by the increasing gap between processor speeds and memory latencies. While processor speeds increase at a rate of approximately 60% per year, memory latencies are reduced by only 7% per year. We are investigating both software techniques and hardware enhancements to help reduce the gap. The work reported here focuses on methods for extracting *partial data traces* during the execution of an application in order to later analyze these memory footprints and alleviate memory bottlenecks through program transformation or hardware reconfiguration.

The paper is structured as follows. We first introduce a method for extracting partial data traces. Next, we develop a hierarchy of compact representations of traces for regular accesses, including an efficient online algorithm for detecting regular accesses. We also provide an order-preserving abstraction of the partial data trace. Our preliminary results show the effectiveness of our techniques to compactly represent data traces. We then discuss several applications of partial data traces. We contrast our approach with prior work. Finally, we summarize our contributions.

2. Partial Data Traces through Dynamic Binary Rewriting

Partial data traces represent 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. Thus, the instrumentation captures the data trace of the 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.

DynInst [3], a component middleware that was designed primarily for "debugging, performance monitoring and application composition out of existing packages", provided the fundamental software infrastructure. While traditional debugging and performance monitoring approaches insert instrumentation at compile time, at link time or at post-link time, DynInst *dynamically modifies a running application* in order to insert instrumentation snippets. However, DynInst Version 2.3 is currently constrained to provide instrumentation only at subroutine calls, entries and exits. Furthermore, its design assumes a dual process model depicted in Figure 1: A control process attaches to an application process to control the application's state (to suspend or resume execution) and to modify the application itself, *e.g.*, by inserting or removing instrumentation code. This model entails a considerable overhead due to system calls and process context switching, both of which also perturb the application's behavior. For example, caches become dirty or may even be flushed due to the execution of the control process or kernel code during system calls and context switches. Upon resuming the application, cold misses may be incurred that would have been avoided had the execution not been suspended.

We have extended the capabilities of DynInst with a set of techniques to support advanced performance monitoring, including partial memory tracing. This was accomplished through a sequence of improvements to the software infrastructure.

First, we demonstrate the capability for extracting the data references of a running application. For this purpose, instrumentation on a per-instruction basis is required, conditioned by the type of instruction, such as a load or store in the case of memory tracing. Instrumentation may be placed in selected subroutines or throughout the entire program. The instrumentation consists of a breakpoint at each load or store instruction in the application, at which point the control process may evaluate the address reference as discussed in the following steps. Second, instructions are decoded to infer the registers involved in address translation of data references. Third, the address of a data reference is calculated based on probed register values and

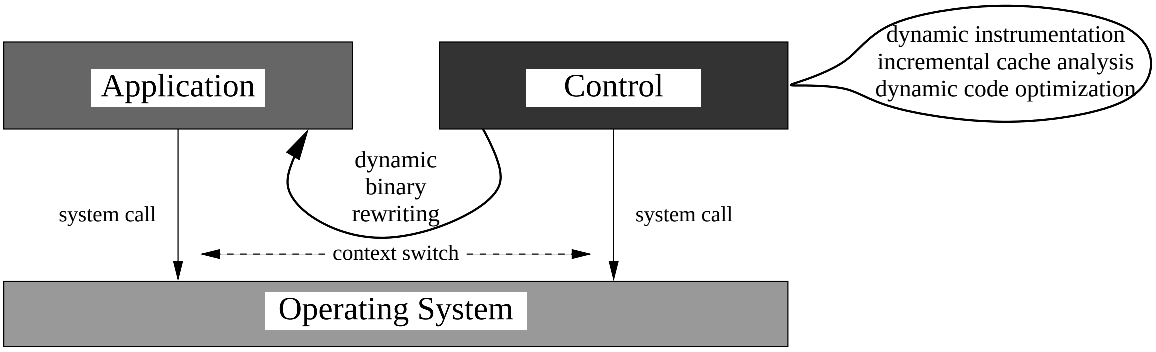

Figure 1: Dual Process Approach to Dynamic Binary Rewriting with Context Switch per Instrumentation Point

address translation rules. This approach suffers from a considerable performance overhead due to the dual process approach discussed previously.

Our initial improvement addresses these shortcomings by extending the framework. First, instruction instrumentation replaces breakpoints with native instrumentation code that avoids system calls. Second, scratch registers saved by the initial trampoline of DynInst are made accessible from within the running application through extended runtime support. Third, non-scratch registers are saved by a second trampoline in the instrumentation. Fourth, we extend the current version of DynInst to include a generic hook to call an arbitrary user-specified subroutine. When used in conjunction with dynamic loading of shared libraries, a feature already supported by DynInst, this generic hook allows the invocation of an arbitrary subroutine from a shared library not present at link time. This differs from the load library feature of DynInst in two ways. We do not parse the structure of loaded modules while DynInst does and we provide additional calling context while DynInst does not. Fifth, data references are translated to generate addresses based on scratch and non-scratch registers by the subroutine. This process is depicted in Figure 2. After the initial instrumentation, partial address traces are generated through repeated invocations of the probe snippets during the execution of the application, *i.e.*, without involving any interaction between the control process and the application, in particular, without the overhead of context switching. This is reflected in the figure by the absence of the control process.

The generation of partial address traces provides the capability to later analyze this trace. The generated partial trace is still potentially very large, a problem addressed in the next section.

3. Recognition of Regular Accesses

This section focuses on the development of techniques to efficiently recognize and compress address trace patterns. Regular access patterns to arrays often occur in tight loops and are not necessarily constrained to numerical applications. These patterns can be represented via *regular section descriptors (RSDs)* [13] as a tuple of <start address, length, stride, access type> [1] as depicted in Table 1. The access type al-

RSD:	<start, length, stride, access type> access type is Read or Write
PRSD:	<start, length, stride, PRSD2> PRSD2 is an RSD or PRSD of the repeated subset
DS:	<DS1, DS2, IV> DS1 and DS2 are a DS, RSD or PRSD; DS1 and DS2 are the primary and secondary substream, respectively
IV:	<pp, pk, sk> with length parameters pp (primary prologue), sk (secondary kernel) and pk (primary kernel)

Table 1: Abstract Pattern Representations

lows the distinction between read and write references, which may be useful in assessing allocation policies in cache simulations. 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

[1] Havlak and Kennedy actually use a stop address instead of the length parameter, which is equivalent. However, they omitted the access type.

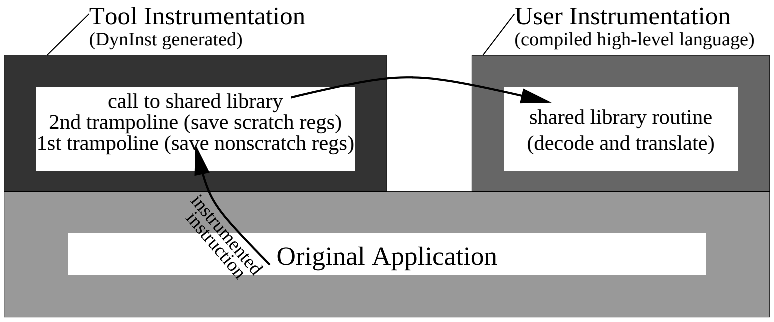

Figure 2: Partial Data Traces without Context Switches

can be represented as RSDs with a constant stride of zero. Consider the example with a row-major layout in Figure 3. 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. This can be represented as RSD4:<B+n+1, n-1, n, Read>. For array A, n-1 read accesses occur at offset 0, captured by a zero stride as RSD1:<A, n-1, 0, Read>. Similarly, the write accesses to offset 0 are captured by RSD3.

```
// declare A[n], B[n][n], initialize A with 0
FOR i := 0 TO n-2 DO
  FOR k := 0 TO n-2 DO
    A[i] := A[i] + B[i+1][k+1];
```

references:
A[0] B[1,1] A[0] A[0] B[1,2] A[0] A[0] B[1,3] A[0] ...
A[1] B[2,1] A[1] A[1] B[2,2] A[1] A[1] B[2,3] A[1] ...

offsets within A:	stream representation:	
reads: 0 0 0 ...	RSD1:	<A, n-1, 0, Read>
1 1 1 ...	RSD2:	<A+1, n-1, 0, Read>
	PRSD1:	<A, n-1, 1, RSD1>
writes: 0 0 0 ...	RSD3:	<A, n-1, 0, Write>
1 1 1 ...	PRSD3:	<A, n-1, 1, RSD3>
	IV3:	<1, 1, 1>
data stream for A	DS1:	<PRSD1, PRSD3, IV3>
offsets within B (reads only):		
n+1 n+2 n+3 ...	RSD4:	<B+n+1, n-1, 1, Read>
2n+1 2n+2 2n+3 ...	PRSD4:	<B+n+1, n-1, n, RSD3>
	IV4:	<1, 2, 1>
overall data stream	DS2:	<DS1, PRSD4, IV4>

Figure 3: Handling Regular Data References

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 Table 1. 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.

The example in Figure 3 illustrates how all read accesses to array A can be combined in PRSD1:<A, n-1, 1, RSD1>. A total of n-1 repetitions of RSD1 with increments of stride one between base addresses of RSD1 are represented. The write accesses to A and the read accesses to B are represented similarly to PRSD3 and PRSD4, respectively.

As illustrated by the example, PRSDs provide a much more compact representation than RSDs. Past efforts to compactly represent access patterns based on RSDs were generally constrained to simple array accesses, but were neither applicable to stack or heap allocated structures, nor to objects [13]. PRSDs actually provide the means to compactly represent these when the padding between stack or heap data structures or objects is regular. For example, a set of objects allocated on the heap may be accessed locally by member variables as well as by a linked list between objects. PRSDs can be used to represent a repeating sequence of regular accesses at both the level of member variables and objects if the objects are located at evenly spaced addresses. Consecutive requests to a memory allocator will provide evenly spaced addresses. In fact, as long as the requests to the memory allocator follow a regular pattern, the objects will be located appropriately. Thus, we expect our techniques to apply to many pointer-based applications.

4. Ordering of Accesses

The previous section provided compact representations for regular access patterns within a sequence of data references. Data reference streams in numerical codes often exhibit accesses to multiple sequences in an interleaved manner. Consider the example in Figure 3 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 *data stream (DS)*. The DS extends a primary stream (PRSD or DS) by an interleaved secondary stream (PRSD or DS) with an *interleave vector (IV)* (see Table 1). The secondary PRSD relates to its primary counterpart through the IV, which is represented by three length parameters `IV<pp,pk,sk>`. The *primary prologue length (pp)* specifies the number of primary references before a secondary reference is issued. The *primary kernel length (pk)* and the *secondary kernel length (sk)* refer to the number of references of each sequence between alternations, respectively. The DS may be hierarchically structured, *i.e.*, a primary data stream may itself contain a secondary data stream instead of a PRSD. The interleave vector then indicates the prologue of the primary stream, followed by alternating references of the secondary and primary kernel lengths from the respective streams, which allows references to interleave at different frequencies. Non-constant functions of strides in RSDs would also require equivalent functions for the length components of IVs, which are not considered in this paper. The example in Figure 3 has a data stream DS1 described by the interleaving of PRSD1 and PRSD3 with an interleave vector IV3. The interleave vector indicates that one initial element from `A` leads the stream (pp=1). Within the kernel, alternations between one secondary element (sk=1) and one primary element (pk=1) follow. The stream DS2 specifies the interleaving between DS1 and PRSD4 with vector IV4. The primary kernel of IV4 has a length of two elements (pk=2) since a write to `A` is followed by a read to `A` from DS1. Notice that multiple PRSDs may be interleaved within a hierarchy of data streams. The example could be extended by a read access to yet another array `C` within the inner loop. This would result in another stream embedding accesses to `C` within DS2 with an interleave vector of `<2,3,1>`.

So far, we have only addressed regular access patterns and their representation. Data streams are powerful enough to represent irregular accesses as well. Each irregular access could be represented as a single RSD. A more efficient representation, however, may be via *irregular access descriptors (IADs)* of the form `<address, i>`, where the i^{th} reference at a given address within the overall data stream is specified. A vector of IADs may then represent all irregular accesses. Access types could be distinguished by keeping separate read and write vectors for IADs.

This abstraction of a data stream suffices to provide a compact representation of regular references within applications. The remainder of this paper discusses the use of data streams, as well as benefits of the overall infrastructure for other applications.

5. Online Detection of Regular Streams

In this section, we present an efficient *online* algorithm — both in terms of space and time complexity — to detect streams by performing an analysis on the data trace. Our online stream detection algorithm assumes that RSDs within a stream are solely comprised of access patterns with constant strides. We allow members forming an RSD to be non-consecutive in the original issuing sequence of the program. This is necessary for dealing with real programs, in which accesses to local stack variables are interspersed among stream references.

We would like our algorithm to detect the RSD corresponding to accesses to a data structure such as an array, despite the interleaving of alternate accesses to other data. Constraints on space require us to periodically discard the older trace data to make room for newer references. We refer to this concept as *aging.* While aging does reduce the possibility of detecting very widely spaced RSDs, patterns with many intervening accesses are less likely to contribute toward temporal locality (or even spatial locality). Hence, aging can be employed to generate irregular access descriptors (IADs), as discussed above.

The algorithm requires maintaining two separate data structures:

- **Stream Table**: This data structure contains a compact description of RSDs that have already been detected. The table is stored as a chained hash with the *expected successor reference address to the RSD* serving as the hash key. Each node in the table is an RSD, *i.e.*, a tuple consisting of the start address, the length, the stride and an access type. If aging of RSDs is desired, an additional value representing the *age* of the last referenced RSD element is added to the tuple. An additional tuple field implements chaining in the hash table.
- **Pool**: This data structure contains the references that have not yet been identified as part of any RSD. 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 the pool, and, consequently, older references are aged and promoted

```
WHILE new reference exists DO
  Increment column; /* move window */
  pool[0][column] := new reference; /* Add reference 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 window size DO
      pool[i][column] := pool[0][column] - pool[0][column-i];
    END FOR;
    found := FALSE;
    /* Search for RSDs of minimum length 3 */
    IF there exists i in 1..size AND k in 1..w THEN
       Enter RSD in stream table;
       Mark columns 0,i,k in pool (shaded in example);
    END IF;
  END IF;
END WHILE;
```

Figure 4: Online Algorithm to Detect RSDs

to the corresponding stream of IADs. In order to determine the existence of RSDs with constant strides, it is imperative to compute differences between elements of the pool. To reduce the computational complexity in repeatedly determining the differences between existing elements as new elements are added to the pool, we keep track of differences with prior elements by storing 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 consecutive stream elements are identical. Practically, 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 4. We illustrate the application of the algorithm on the example in Figure 3. We assume `A` and `B` start at location `100` and `200`, respectively, and are stored in row-major layout. For simplicity, we assume both `A` and `B` have 10 elements each, and each element occupies a single memory location. The accesses translate into an address sequence as follows:

100 ; 211 100 100 ; 212 100 100 ; 213 100 100 ; ...
101 ; 221 101 101 ; 222 101 101 ; 223 101 101 ; ...

Figure 5 shows the snapshot of the pool as the first eight references are encountered. The header row

dist.	100	211	100	100	212	100	100	213
-1		111	-111	0	--			--
-2			0	-111	--			--
-3				0	1			1
-4								--
-5								--
-6								1

Figure 5: Snapshot of the Reservation Pool

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 4). The particular element used for calculating the *difference* depends on the row in which the difference is computed. The first row (below the header) consists of the difference between the current and the immediately preceding element (distance -1), exemplified by the upper arrow. The second row consists of differences between the current element and its second predecessor (distance -2), illustrated by the lower arrow, and so forth. To capture RSDs within a window size `w`, we need only compute the differences above the diagonal of the pool table. Elements determined to be part of a stride are removed from the table. In the example above, on seeing the third `100` (assuming a minimum length of three), we will identify an RSD by observing the two corresponding differences of `0` (circled) in a transitive relationship. Consequently, we will insert an RSD of `<100, 3, 0>` in the stream table. These elements are shown shaded in the pool to illustrate their absence from the subsequent difference computations of the pool. Similarly, the later `100`s will be immediately observed to belong to the RSD, and the RSD fields will be modified to `<100, 5, 0>` on receiving the fifth `100`. It is important to observe that on detecting an element to be a part of an RSD, we still keep a slot for it while omitting differences for this element. The slot is kept to preserve the notion of the *window size*. On seeing `213`, a new RSD is identified by observing an identical difference of `1` (circled) for the transitive relation between 211, 212 and 213. At this point, `<211, 3, 1>` will be inserted in the stream table, and the slots for these elements can be marked to indicate their non-participation in further RSD detection.

We can summarize with the following observations:

- There is an implicit assumption that sequences must have a minimum length, at which point the corresponding accesses are promoted to an RSD. This detail is omitted from the pseudo-code representation of the algorithm. In the example, this value is three.
- The worst case complexity of the algorithm is $O(N \times w^2)$, where N is the number of total reference and w is the window size. This can be significantly reduced if the differences with prior elements are not computed when it becomes clear that no stream with the minimum length can be found in the window.
- Our stream table is optimized for the average case, where an element does not belong to any RSD. Extending RSDs in our hash is more expensive than performing an unsuccessful lookup, since it involves changing the hash bucket of the RSD.
- Our algorithm intentionally prevents membership of an element in multiple streams. We achieve this by removing elements from the pool on detecting their membership in an RSD.
- As mentioned before, aging of streams can easily be achieved by including a tag with each tuple in the stream table signifying the stream's age.

We omit the details of composing RSDs into PRSDs and data streams (DS), since these tasks do not present a significant contribution to the algorithm.

6. Preliminary Results

Preliminary results of our study were conducted on the Power 3 architecture clocked at 222 MHz under AIX 4.3 for a set of kernels with nested loops accessing matrices. Our initial instrumentation with two context switches resulted in an overhead of about one second per instrumentation point. By implementing inline instrumentation as described in Section 2, this overhead was reduced by three orders of a magnitude to about 560 μsecs per instrumentation point. The test kernels can be described as follows. `Modulo` performs a wrap-around modulo index into a matrix, `MM` is a regular matrix multiplication and `tiled MM` is a tiled version with a blocking size B=50 resulting in 50×50 tiles for an $N \times N$ matrix. Table 2 depicts the number of RSDs and PRSDs for the central loop of each program. Notice that we only depict the metrics of *one* array access. There were a total of three read accesses and one write access per iteration for each kernel. For `Modulo` and `MM`, a single RSD suffices to represent the regular accesses per array for the innermost loop. `Tiled MM` required B separate RSDs for each blocked loop since repeated accesses to each block are issued via separate blocked streams. `Modulo` required one PRSD to represent the modulo arithmetic in indexing the matrix. `MM` required two PRSDs for the middle and outer loops of the matrix multiplication depicted in Figure 6. `Tiled MM` required three PRSDs with B variations, depicted for N=1000, B=50 and a data size of 8 bytes per reference in Figure 6. One PRSD in addition to the two PRSDs of `MM` captures the blocking for one dimension. The second dimension requires separate modeling for each blocked stream resulting in a total of B three-level PRSDs with one RSD each. Figure 6 depicts two sets

program	Modulo	MM	tiled MM
RSDs	1	1	B (50)
PRSDs	1	2	3×B (150)

Table 2: Number of Descriptor per Kernel

```
MM,         PRSD1: <     &B,    0, 1000, PRSD2>
N=1000:     PRSD2: <     &B,    8, 1000,  RSD>
              RSD: <     &B,    0, 1000, Read>

tiled MM,   PRSD1: <     &B,    0,   50, PRSD2>
N=1000,     PRSD2: <     &B, 8000, 1000, PRSD3>
B=50:       PRSD3: <     &B,    8,   50,  RSD1>
             RSD1: <     &B,    0,   50, Read>

            PRSD4: <&B+400,    0,   50, PRSD5>
            PRSD5: <&B+400, 8000, 1000, PRSD6>
            PRSD6: <&B+400,    8,   50,  RSD2>
             RSD2: <&B+400,    0,   50, Read>
```

Figure 6: Descriptors: One for MM and two (out of 50) for Tiled MM

of such 4-level descriptors. We omitted one additional RSD (and the corresponding PRSDs) from the table, which results from starting the partial data trace in the middle of an iteration and results in an RSD with shorter length than observed during consecutive iterations. Overall, the results confirm that regular accesses can be represented in a compact manner. For two kernels, the compaction factor is $N \times N$, *i.e.*, the resulting access representations are of constant size. For the last kernel, the compaction factor is $N \times N/B$. In practice, the blocking of tiled matrix multiply is a constant implying that the order of compaction is still $N \times N$.

7. Applications of Partial Data Traces

We have demonstrated the ability to extract data references from binaries and have established methods to represent data streams of references in a compact manner. The compression of data streams is integrated into the instrumentation of the binary to avoid the generation of voluminous traces. These compressed traces may be communicated on demand as partial traces to another process, such as the control process.

7.1. Incremental Cache Analysis

In the case of cache analysis, the cache behavior is simulated incrementally based on the partial memory traces as they are supplied. Cache simulation allows the identification of the causes of cache misses, such as cold misses, conflict misses and capacity misses. Only the latter two are relevant for the programmer since only they may be avoided. The cache simulator provides the means to track the sources of conflicts, *i.e.*, a cached data item replaced by a miss is recorded in conjunction with the miss. The simulation results can be depicted with a reference to the source program, thereby guiding the programmer to hot spots of cache misses. Conflict misses correlate the sources of a miss with the item replaced in cache. Capacity misses may be regarded as a special case of conflict misses where a data structure *conflicts with itself*. A correlation between conflicting items on the level of data structures provides sufficient information to the programmer to help restructure the application program and, subsequently, to avoid such a conflict. Application restructuring can yield considerable performance gains. Overall, incremental cache simulation and visualization of cache correlations identifies memory bottlenecks (hot spots of data misses). This information enables the programmer to restructure the data layout or the iteration structure over the data space in question.

7.2. Dynamic Code Optimization

Information about the cause of cache misses may also be exploited by dynamic code optimizations. Software prefetching can be used in conjunction with loop unrolling to selectively prefetch data where cache misses are known to occur regularly. More complex optimizations, such as tiling and other loop transformations, may be applicable but are subject to data dependence constraints that can be inferred from data flow analysis [28, 29]. Applying such transformations dynamically has considerable advantages over compile-time optimizations. During execution, the architectural parameters, such as cache size, are known, and this knowledge may result in program transformations better geared toward a particular architecture. We can restrict these optimizations to hot paths that may be detected by instrumentation along the lines of portable frameworks that target different processors, such as UQBT [9, 26]. Our infrastructure permits these optimizations to occur offline before the optimized code is injected into the application. The application may proceed to execute while the binary is being optimized, which reduces the overhead of dynamic compilation typically imposed by just-in-time compilation.

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 [23]. The performance improvements of three orders of a magnitude 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, Havlak and Kennedy's RSDs [13] are virtually identical to the *stream descriptors* in use at about the same time in the compiler and memory systems work inspired by the WM architecture [30].

Atom has been widely used as a binary rewriting tool to statically insert instrumentation code into application binaries [25]. 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.* [27] describe an analytic framework for the evaluation of 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 and is used to describe the state of a caching system. All such notations support the creation of tools for automatic trace expansion or synthetic trace generation, and can be used to represent different levels of abstraction in benchmark analysis.

Buck and Hollingsworth performed a simulation study to pinpoint the hot spots of cache misses based on hardware support for data trace generation [4]. 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, either 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, 22].

Recent work by Mellor-Crummey *et al.* uses a modified compiler 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 [21]. 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 [8, 7] as well as quantitative metrics to assess memory bottlenecks within the data reference stream [6]. 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 [3, 14] using cache miss equations [11] or symbolic reference analysis at the source level based on Presburger formulas [5]. 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 [24, 2, 9, 26, 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.

9. Conclusion

We introduced an approach to dynamic binary rewriting and motivated its benefits for identifying cache performance bottlenecks and for applying dynamic code optimizations. We developed a framework to extract partial data traces in a portable fashion from uninstrumented executables, and contributed methods for compactly representing these traces. An online algorithm was presented to capture regular access patterns efficiently through regular section descriptors. A hierarchical representation, power regular section descriptors (PRSDs), extends this notion to capture recurring patterns with different base addresses, and the abstraction of data streams provides an ordering for the interleaving of different PRSDs. Preliminary results show performance improvements of three orders of a magnitude of inline instrumentation over a dual process approach involving context switching. We also report constant size representations for regular access patters in nested loops. We are currently pursuing several directions to exploit the knowledge of data streams in the context of software optimizations and, potentially, specialized hardware support.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[2] 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.

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

[4] Bryan R. Buck and Jeffrey K. Hollingsworth. Using hardware performance monitors to isolate memory bottlenecks. In ACM, editor, *SC2000: High Performance Networking and Computing. Dallas Convention Center, Dallas, TX, USA, November 4–10, 2000*, pages 64–65, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2000. ACM Press and IEEE Computer Society Press.

[5] 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.

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

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

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

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

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

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

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

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

[15] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. Informing memory operations : Providing memory performance feedback in modern processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecure*, pages 260–270, New York, May 22–24 1996. ACM Press.

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

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

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

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

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

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

[22] T. C. Mowry and C.-K. Luk. Predicting data cache misses in non-numeric applications through correlation profiling. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-30)*, pages 314–320, Los Alamitos, December 1–3 1997. IEEE Computer Society.

[23] 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, September 2000.

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

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

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

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

[28] M. Wolf and M. Lam. A data locality optimizating algorithm. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44, June 1981.

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

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