A Tool for Characterizing and Succinctly Representing the Data Access Patterns of Applications

Catherine Mills and Allan Snavely
Department of Computer Science and Engineering
University of California at San Diego
Email: (crmills,allans)@cs.ucsd.edu,(cmills,allans)@sdsc.edu
Laura Carrington
San Diego Supercomputer Center
Email: lcarring@sdsc.edu

***Abstract*—Application address streams contain a wealth of information that can be used to characterize the behavior of applications. However, the collection and handling of address streams is complicated by their size and the cost of collecting them. We present PSnAP, a compression scheme specifically designed for capturing the fine-grained patterns that occur in well structured, memory intensive, high performance computing applications. PSnAP profiles are human readable and reveal a great deal of information about the application memory behavior. In addition to providing insight to application behavior the profiles can be used to replay a proxy synthetic address stream for analysis. We demonstrate that the synthetic address streams mimic very closely the behavior of the originals.**

I. Introduction

The memory address stream of an application contains a wealth of information which can be used to characterize the memory behavior of the application. High performance computing (HPC) application are often memory bound, due to this and the effects of the "Von Neumann Bottleneck," the memory behavior of an application often dominates the overall performance. Memory address streams can therefore be used to drive trace-driven memory simulations to explore compute system and application software improvements [1], [2], [3], [4], [5].

Recently, power consumption patterns have been shown to have a very high correlation with memory access patterns [6] opening up another pivotal area of research dependent on address stream traces. Power consumption has been identified as a primary challenge to exascale computing [7].

Despite the general usefulness of trace-driven memory simulation the collection, handling and storage of memory address traces remains problematic. Address streams for large-scale high performance computing (HPC) applications are more problematic than most due to large numbers of processors and the expense of using large resources. Challenges associated with address trace collection include space costs, time costs, accessibility, and proxy inaccuracies.

Space Costs: Address streams are extremely large. It is possible for an application's address stream to grow faster than 2.6 TB per hour per core [8]. HPC applications are run on thousands of processors and potentially for many hours, further exacerbating the issue.

Time Costs: Collecting address streams and storing them to disk, as well as retrieving them from disk, is slow. The processing rate for an address stream that is being read from disk is limited by disk speed. Optimistically, this would be 250 MB/s (addresses can be generated faster than 700 MB/s).

On-the-fly processing of address streams bypasses the space challenges by never storing the trace. However, this processing must be done on an HPC resource and causes at least a 10X slowdown even with aggressive sampling techniques. Without sampling, the slowdown is much worse, 100X-1000X [9]. Additionally, the process must be repeated any time the experimental parameters change, requiring further use of the HPC resource.

Accessibility: Raw address streams themselves provide little if any insight to code structure or access patterns. A stored address stream is too large to be read and understood by a person.

Proxy Inaccuracies: One common approach to avoiding the use of large address streams is to work with application kernels or representative benchmarks, rather than real workloads. However, it is very difficult to find a benchmark that can act as an accurate proxy for HPC applications. The validity of a simulation depends heavily on the chosen input workload [10], [11]. The performance results obtained by traces of small benchmarks chosen to represent a high performance computing (HPC) workload are of questionable relevance during such evaluations; choosing appropriate benchmarks is a difficult task, especially when applied to an HPC workload [12].

Obtaining and storing relevant address traces is a fundamental requirement for trace-driven memory simulation of large-scale HPC applications and the question must be asked: *how does one provide valid and relevant input of substantial size to a simulation?*

This paper presents Synthetic Address Stream Profiles

(PSnAP). PSnAP is a compression technique designed specifically for HPC address streams. It takes advantage of patterns found in the per instruction address streams of an application and creates very small profiles. These profiles can be used to replay the address stream. PSnAP resolves many of the difficulties associated with address stream collection including space costs, time costs, accessibility, and proxy inaccuracies.

Space Costs: PSnAP profiles are extremely compact, in the range of kilobytes, meaning they can be emailed between collaborators. An important characteristic of the profiles is that they grow as a function of the complexity of the source code structure rather than dynamic execution time.

Time Costs: Once collected PSnAP profiles can be reused repeatedly without the use of an HPC system. The slowdown incurred by tracing for PSnAP is only incurred a single time and tracing can be performed for specific areas of an application, further reducing HPC system use. The replay times for PSnAP are fast enough that they can be replayed at least at the rate of disk speeds.

Accessibility: The PSnAP profiles are human readable and manipulatable. High level structures and fine-grained patterns can be seen in the profiles, application phases can be identified, and the results of compiler optimizations such as loop-unrolling and function inlining can be seen.

Proxy Inaccuracies: There is no need to use benchmark kernels to represent HPC application address streams; the actual streams can be compressed.

The contributions of this work are as follows:

- A new per-instruction recording technique that captures access patterns in a concise manner.
- A human readable address stream profile that can be manipulated to evaluate different auto-tuning strategies.
- A synthetic stream generation method that leverages existing control flow compression and per-instruction address streams.

We evaluate PSnAP based on accuracy and performance using the NAS Parallel Benchmarks [13]. Accuracy must be examined because PSnAP uses lossy techniques. We compare the observed and synthetic address streams using cache hit rates. PSnAP achieves accuracy in the full execution measurements, on average the difference between observed and synthetic streams is .08%.

PSnAP's performance includes, execution overhead, profile size and replay time. A comparison of these values shows that PSnAP beats the state of the art in all three categories.

In addition to good performance, the format of PSnAP profiles allows for an array of new usage models. For example, PSnAP profiles can be partially replayed for examination of specific loops, can be replayed in chunks in order to fit into memory, and can be manipulated for experimentation.

II. PSnAP Approach

PSnAP profiles are generated by running an instrumented executable on an HPC resource. The address stream is captured on-the-fly and a minimal amount of information is saved to raw PSnAP profiles. The raw PSnAP profiles are processed further off-line. The post-processing does not require the use of an HPC resource.

The instrumentation is done using a binary rewriter, PEBIL [14]. The instrumentation tool collects address tuples into a buffer and, when the buffer is full, passes the set of tuples to the PSnAP library. PEBIL is an open source lightweight binary instrumentation tool that can be used to capture information about the behavior of a running executable. It works on the Linux/X86 instruction set.

This section describes our approach to representing instruction address streams. We begin with an overview of the representation and then describe the methods of recording, decoding and replaying address streams.

A. Stream Representation

Three core concepts guide this compression technique. This section describes the motivation behind each concept using an example instruction taken from the most dominant[1] basic block from the NAS Parallel Benchmark FT [13].

1) Address streams generated per instruction are simpler than per basic block, loop, or application.
2) Any instruction address stream can be described in terms of a starting address and a stride pattern.
3) Describing an address stream in terms of strides is often more succinct than describing it in terms of addresses.

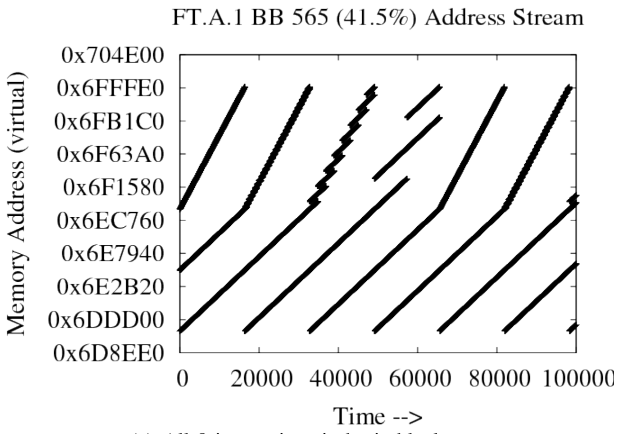
Address streams generated per instruction are simpler than those generated per basic block, loop, or application. This level of granularity was chosen because patterns within small regions of memory are often simple and easily compressed [15]. In most cases, instructions act on a contigious region of memory.

A simple example demonstrates the benefits of focusing on per instruction address streams. Figure 1(a) shows the address stream for the entire dominant basic block of FT. The virtual address space is covered by the y-axis. Time is represented along the x-axis (one address per unit time). There are two visible patterns in the stream, the upper half has four smaller patterns that are repeated and the lower half is a single simple pattern. Contrast this to Figure 1(b) that shows the address stream for only the first instruction of the block (out of 8). It covers a smaller area (address space) and presents a much more simple pattern.

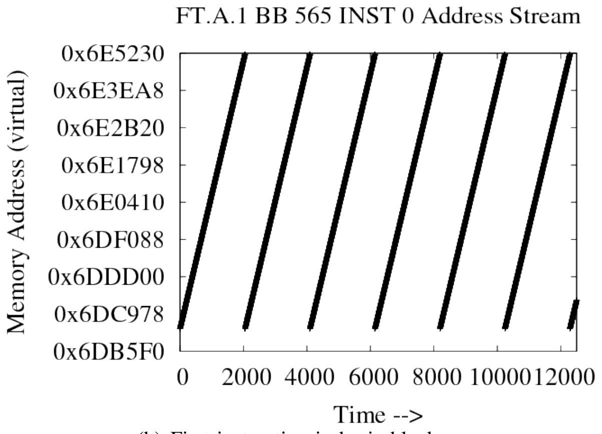
The pattern seen in Figure 1(b) can be succinctly represented using strides[2]. The range of addresses produced is fairly large, but over the entire execution only four strides are encountered. Figure 2(a) shows the same range of time, but plots the strides rather than instructions. In this figure is appears that there are 2047 small strides (the thick line across the top) followed by one large backward stride (size 36,816 bytes). Figure 2(b) is a zoomed in view of the first 48 strides in this stream (the thick black line in Figure 2(a); this view reveals that there are 15 strides of size 16 followed by a stride

[1]Basic block dominance is determined by the number of memory instructions executed on behalf of each block over an entire run.

[2]A stride is the numerical difference between two subsequent addresses.

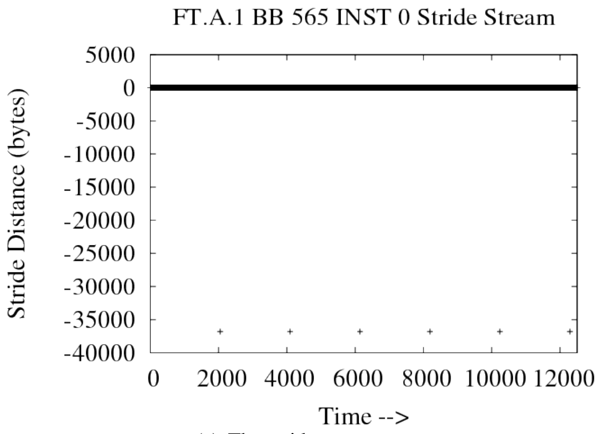


(a) All 8 instructions in basic block.

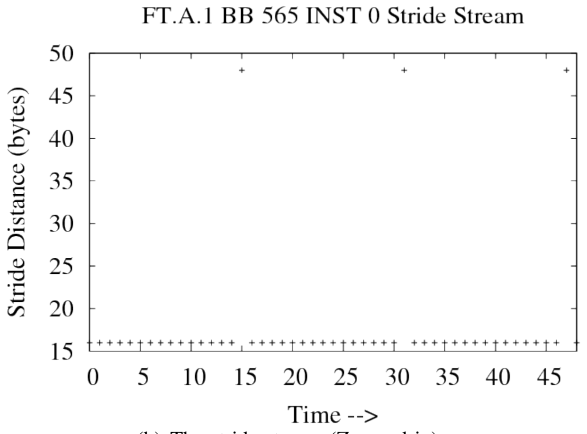


(b) First instruction in basic block.

Fig. 1. Address Streams: Dominant block in FT.



(a) The stride stream.



(b) The stride stream (Zoomed in).

Fig. 2. Reducing an address stream to strides allows for a much more succinct representation of the stream.

of size 48. We refer to this repeated pattern as the instruction's *stride pattern*.

Any instruction address stream can be described in terms of a starting address and a stride pattern. Continuing with the current example, a manual inspection of the stride pattern reveals a fixed stride pattern that can be expressed as an expression (very much like a regular expression) where A, B, C, and D are the recorded strides and the exponents indicate the number of times to repeat a pattern.

$$((A^{15}B)^{127}A^{15}C)^{16384}(A^{15}B)^{127}A^{15}D \qquad (1)$$
$$where: A = 16, B = 48, C = -36816, D = -18384$$

Describing an address stream in terms of strides is often more succinct than describing it in terms of addresses. In this example it is possible to express the first phase of this address stream using less than 70 bytes. The first phase of this address stream represents approximately 23 Mb of addresses, this is a compression of over 300,000 times. A large portion of address streams are well represented in this manner. We show that out of over 13,000 instructions in the NPBs, only 29 fail to fit this model and, therefore, require a different representation. Not only is there a high compression rate, but the address stream pattern has been encapsulated in a way that is easily accessible to the reader.

The amount of space required for this representation depends directly upon the number of unique strides that occur in the address stream. For instance, in a random address stream, one that may result from a gather operation, there are as many strides as there are addresses. This type of address stream is not a candidate for this representation. It is identified as such (early in instrumentation) and alternate means of recording are employed.

B. Recording and Decoding Stride Patterns

The following describes how the stride patterns are determined from a raw address stream. The PSnAP library receieves a stream of address tuples from the instrumentation code. Each tuple contains an identifiers for the basic block and instruction as well as the memory address. After recording the first address encountered by a specific instruction the algorithm focuses on strides.

The stride patterns are recorded by counting occurrences of unique strides. When a new stride is encountered it is put at the end of an array list ($strides$) at index i. Another array $count$

is updated so that $count(i) = 1$ indicating that $stride(i)$ has been encountered once. In addition, each time a new stride is encountered a snapshot of the count array is recorded.

The cost of updating the arrays is linear with respect to the number of unique strides. The number of uniques strides for a specific instruction is quite small, during our experiments no instruction had more than 15 unique strides associated with it. Furthermore, in the examples shown in this section the matches that occur at the first index make up 94% of the total accesses.

Table I shows an example of this data. There are four strides recorded. The first time that a stride of size 48 is encountered, a stride of size 16 has already been encountered 15 times. This can be seen in the first row of the stride history in Table I. At the end of execution the snapshots contain the values needed to create the expression in Eq. 2 along the diagonal. The series of snapshots is referred to as a *stride history*.

TABLE I
STRIDE HISTORY FOR THE FIRST INSTRUCTION OF THE DOMINANT BLOCK IN FT.

Stride	16	48	-36816	-18384
Count	377487360	24903680	131071	131072
stride history				
1	15	1	0	0
2	1920	127	1	0
3	31458240	2080831	16384	1

This example illustrates the basic idea behind the PSnAP approach. *The repeated stride pattern can be found by examining a stride history.*

After the stride histories have been recorded they must be decoded in order to prepare for replay. Decoding refers to the process of transforming a stride history into a stride pattern. The decoding is done off-line, meaning that it does not require the use of an HPC resource.

Decoding involves three steps. First, each instruction is categorized(*identification*). Second, transformations are performed on the stride histories to reveal the stride patterns(*reduction*). Lastly, the stride patterns are used to guide address stream reconstruction(*replay*).

Identification involves classifying each stride history. The following pattern classifications are defined: constant, simple repeat, simple alternating, complex repeat, and undesignated.

The *simple repeat*, *simple alternating* and *complex repeat* classifications result from nested loop patterns moving through multi-dimensional data. Simple repeat is the basis for each of them.

Throughout the discussion on decoding the following definitions apply:

A-Z represent the values in the strides array (in order).

c_i is the value in the i^{th} position of the counts array.

$h_{(i,j)}$ is the value in the i^{th} row and j^{th} column of the stride history.

1) Simple Repeat: A simple repeat pattern results from a nested loop structure stepping through array data. A strided walk through a 1D array is the most basic example. The number of strides collected will depend on the dimension of the data and the depth of the loop structure. Table II shows the data for a simple repeat pattern taken from the FT benchmark.

Identification. Simple repeat patterns are identified using the counts array (row 2 of Table II. Each count in the line must be evenly divisible by the sum of counts that fall after it.

$$\forall i(c_i \% \Sigma_{j=i+1}^{n} c_j = 0) \tag{2}$$

Reduction. Once the instruction has passed the identification test, the next step is to reduce the stride history into representative expressions. The reduction is performed only on the last line of the stride history.

Divide each value by the sum of the values that fall after it in the same row. Given that c_i is the value in position i apply the following.

$$c_i = c_i / \Sigma_{j=i+1}^{n} c_j \tag{3}$$

The repeat value is set to the last value in the counts row. The reduced pattern is therefore: (15,255,2047,1) repeated 7 times.

TABLE II
THE STRIDE HISTORY FOR A SIMPLE REPEAT.

strides	4096	-61424	16	-134217712
counts	62914560	4177920	16376	7
stride history				
1	15	1	0	0
2	3840	255	1	0
3	7864320	522240	2047	1
replay pattern				
pattern	15	255	2047	1

Replay. Replaying the pattern is straight-forward, the pattern is the following.

$$(((A^{15}B)^{255}A^{15}C)^{2047}(A^{15}B)^{255}A^{15}D)^{7} \tag{4}$$
$$A = 4096, B = -61424, C = 16, D = -134217712$$

It appears to be unnecessary to collect the pattern from the last line of the stride history. From Table II it is apparent that the same values appear along the diagonal of the stride history. However, the simple repeat pattern is often embedded within simple alternating and complex repeat patterns. In those situations the full diagonal is not available.

2) Simple Alternating: A simple alternating pattern consists of two or more simple repeat patterns. This can occur during a phase change in the execution or after jumping to a new portion of data that may or may not have the same shape as the previous data. The complex repeat pattern may contain a simple alternating pattern and the method for handling complex repeat can be used for simple alternating.

3) Complex Repeat: The last in this group of patterns is the complex repeat pattern. The complex repeat pattern can comprise both simple repeat and simple alternating patterns. The pattern in the top half of Figure 1(a) is a complex repeat pattern taken from FT. This pattern is found in the fourth instruction of the most dominant basic block. Figure 3 shows the contribution of the fourth instruction to the basic block pattern. Its stride history is presented in Table III[3]. Four distinct patterns can be seen in the first 8000 addresses of the figure. The replay pattern reflects these patterns.

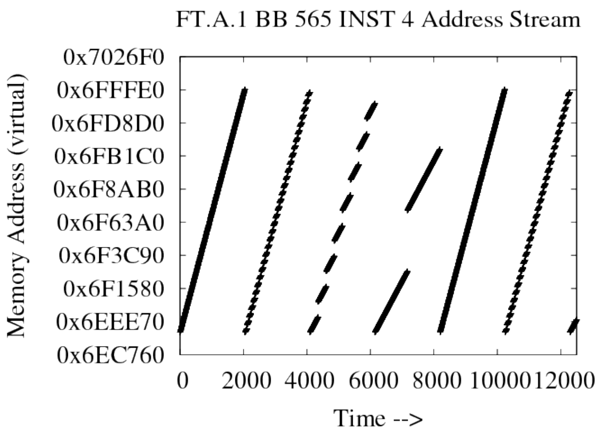


Fig. 3. The address stream for the fourth instruction of the most dominant block in FT.

Identification. The complex repeat pattern is more involved to identify than the previous two examples. The reduction steps must be complete and then each of the resulting patterns are tested for simple repeat and simple alternating.

Reduction. The complex repeat pattern is a combination of other patterns, therefore, the first reduction step is to identify the individual patterns. The *dominant lines* have to be identified. A dominant line is one that describes a complete pattern. Both simple repeat and simple alternating patterns have dominant lines as well, but they are trivial to locate because they are always the last line in the stride history. A dominant line is always followed by a line with a 1 in the diagonal.

Definition 1: $history(i)$ is dominant $\iff history(i+1, i+1) = 1$

Definition 2: $final(i) == j \iff dom(i,j) = 0 \;\&\; \forall_{k=j+1}^{n} dom(i,k) = 0$

Each dominant line is further reduced.

1) Reduce each dominant line by the values in the dominant line directly above it.

$$dom(i,j) = dom(i,j) - dom(i-1,j) \quad (5)$$

2) Subtract one from the position in each dominant line that corresponds to the final value in the dominant line above it.

$$dom(i, final(i-1)) = dom(i, final(i-1)) - 1 \quad (6)$$

The last dominant line is not a simple repeat or simple alternate pattern. This line is referred to as a *master line*. A master line guides the repetitions of the patterns that come before it. This line is reduced to only the entries that correspond with $final$ values in the patterns above it.

$$master(j) = 0 \iff \nexists i s.t. final(i) = j \quad (7)$$

[3]The pattern created by this basic block has two phases (a simple repeat) the second phase has been removed in order to make the example more readable.

Each of the dominant lines is passed through the reduction performed for simple repeat or simple alternating. In this example all of the lines are simple repeat. The reduced dominant lines and final pattern are listed in Table. III.

Replay. The replay for this pattern is performed in the same way that each of the previous patterns. The only addition is that the patterns are grouped together and repeated 4095 times.

$$\begin{aligned}
&(A^{15}B)^{127}A^{15}C \\
&((A^{15}D)^3 A^{15}E)^{31}(A^{15}D)^3 A^{15}F \\
&((A^{15}D)^{15}(A^{15})G)^7((A^{15}D)^{15})^7 A^{15}H \\
&((A^{15}D)^{63}A^{15}I) \\
&((A^{15}D)^{63}A^{15}J) \qquad (8) \\
&where: A = 16, B = 336, C = -73392, D = 48 \\
&E = 1200, F = -72528, G = 4656, H = -69072 \\
&I = 18480, J = -55248
\end{aligned}$$

A representation such as the above is detailed yet compact and is much more human readable than a raw address stream. If needed a human or a post-analysis tool can tell a lot about an application or a section of an application from examining patterns such as the above. We also provide the ability to ”zoom in” on specific loops or functions and show their patterns. Also PSnAP's ease of use allows one to recapture the same stride patterns after a change. For example, a recompile with new flags or a code restructuring, to examine what has changed or improved.

III. Control Flow Compression

As described thus far, PSnAP provides a mechanism to record and replay address streams for individual instructions. A full synthetic address stream requires that those streams be replayed together in the order they were collected. This order is recorded using control flow compression. The control flow compression used by PSnAP follows the method used in Path Grammar Guided Trace Compression (PGGTC) [16] very closely. The following is a brief overview of the approach, highlighting the differences between PSnAP and PGGTC.

The control flow graph of an application is a directed graph that describes the path taken through the code. The application code is broken down into basic blocks, which are represented as nodes in the graph. A basic block is a set of instructions that has a single entry and a single exit.

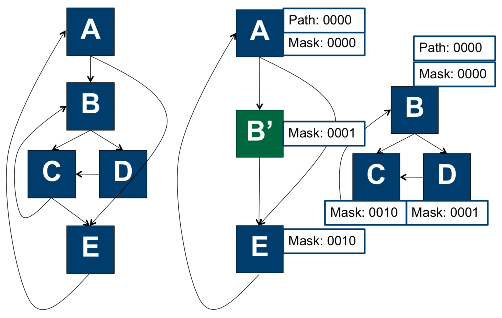
Loop structures and basic blocks are identified statically using a binary rewriting tool PEBIL. The loops and basic blocks are each assigned a unique id. The loop ids are unique over the entire code and the block ids are referred to as masks and are unique only within the containing loop.

Each loop in the application is treated separately. Figure 4(b) demonstrates how sub loops are separated. The entire sub-loop containing blocks B,C and D is moved to a new data structure and is replaced with a dummy basic block. This reduces the complexity of describing the flow through each loop.

Instrumentation code is inserted at the loop heads and at the beginning of each basic block. This code is responsible

TABLE III
THE STRIDE HISTORY FOR A COMPLEX REPEAT.

Stride	16	336	-73392	48	1200	-72528	4656	-69072	18480	-55248	
Counts	377487360	6225920	32768	16809984	1507328	32768	327680	32768	32775	32760	
stride history											
1	15	1	0	0	0	0	0	0	0	0	0
2	1920	127	1	0	0	0	0	0	0	0	0
3	1935	127	1	1	0	0	0	0	0	0	0
4	1980	127	1	3	1	0	0	0	0	0	0
5	3840	127	1	96	31	1	0	0	0	0	0
6	4080	127	1	111	31	1	1	0	0	0	0
7	5760	127	1	216	31	1	7	1	0	0	0
8	6720	127	1	279	31	1	7	1	1	0	0
9	7680	127	1	342	31	1	7	1	1	1	0
10	31457280	520192	4096	1400832	126976	4096	28672	4096	4096	4095	1
reduced dominant lines											
1	1920	127	1	0	0	0	0	0	0	0	0
2	1920	0	0	96	31	1	0	0	0	0	0
3	1920	0	0	120	0	0	7	1	0	0	0
4	960	0	0	63	0	0	0	0	1	0	0
5	960	0	0	63	0	0	0	0	0	1	0
60	0	0	4095	0	0	4095	0	4095	4095	4095	1
replay pattern											
1	15	127	1	0	0	0	0	0	0	0	0
2	15	0	0	3	31	1	0	0	0	0	0
3	15	0	0	15	0	0	7	1	0	0	0
4	15	0	0	63	0	0	0	0	1	0	0
5	15	0	0	63	0	0	0	0	0	1	0
60	0	0	4095	0	0	4095	0	4095	4095	4095	1



(a) The original loop. (b) The resulting loop.

Fig. 4. Loop separation in the control flow graph.

for recording the path taken through the loop and the iteration count for each entry.

The path taken through the loop for a single iteration is expressed in a bit map. The bit map is maintained by the loop head and updated by each visited basic block. Each basic block, or dummy basic block that represents a sub-loop, within the outer loop is statically assigned a mask. Instrumentation code at each basic block performs a bitwise-OR with the loop bit map. At the end of a single iteration of the loop each flipped bit in the map represents a basic block taken.

The ordering of the masks for basic blocks is key for accurate replay. The 1 bits in the bit mask indicate which basic blocks where executed and the order they were executed in. If the basic blocks in figure 4(a) were labeled as B=00, C=01, and D=10 (a valid breadth first ordering) the execution of C and D would be swapped during replay. A topological ordering [17] is used to ensure that the path represents a valid execution ordering of basic blocks.

At the end of each loop iteration the path taken through the loop is compared to the path taken on the previous iteration. If the paths match, a counter is incremented, if not, the new path is pushed on a stack of paths, called the *path history*. A limited number of paths are recorded to save space.

If the number of allowed paths is exceeded, the loop is labeled as having overflowed. A snapshot of the current path history is taken and saved. At that point the history converts to only counting the paths taken, the order is no longer maintained. The number of entries maintained in the history is configurable, for the experiments using the NPBs 20 entries was adequate.

There are several differences between the PSnAP implementation and the original PGCCT implementation. PGCCT is non-lossy whereas PSnAP is lossy. Specifically, rather than collecting new paths indefinitely, PSnAP allows for a finite number of paths to be collected before simply counting unique paths rather than keeping track of the order they appeared in. For example, if two paths were taken through a loop in an alternating manner 1 million times, PGCCT would have 1 million entries in the stack of paths. PSnAP would have only 2, each with an associated count of 500,000. This saves a large amount of space, but results in a potential loss of accuracy, if the pattern cannot be determined. As a hint to the pattern a snapshot of the path history at the point of overflow is saved in the profile.

The other major difference between the implementations is that the bit maps in PGGTC were limited to 32 bits. If the path through a loop contained more than 32 basic blocks a hash map was used. The bit maps in PSnAP have been implemented to expand a byte at a time indefinitely.

The result is a highly compressed trace, even of long running or complex applications or benchmarks, that can be easily shared, even by email, to convey the memory behavior of applications.

IV. Evaluation

The accuracy and performance of PSnAP were evaluated through a series of experiments. The experiments were conducted using the NAS Parallel Benchmarks [13] run on four cores. All of the experiments were run on Dash[4]. Dash is a 64 node system housed at SDSC. Each node is equipped with two quad-core 2.4 GHz Intel Nehalem processors and 48 Gb of memory. The experiments include a coverage survey, manual inspections of key basic block address streams, cache hit rate comparisons, and a performance evaluation.

A survey of instruction address stream characterizations was performed in order to verify that an adequate percentage of instructions qualify to be represented using stride patterns. The accuracy of the synthetic address streams is evaluated in two ways. The instruction-specific streams were directly compared at a basic block level (this implies no control flow). Additionally the cache hit rates of the synthetic address streams were compared to those of the observed address stream. The synthetic address streams prove to be very accurate; the instruction-specific streams are non-lossy with a 100% match rate. Cache hit rates were reproduced with an average error of 0.8% (i.e less than 1%) in L1.

The performance evaluation of PSnAP includes examining the achieved compression rates and the overhead of generating the profiles and synthetic streams. The compression rates are competitive with the state of the art, often surpassing it. At the same time the slowdown is significantly less; PSnAP incurs an average slowdown of 90X. This slowdown is achieved with no sampling and using an unoptimized version of the instrumentation code. The slowdown is already very low for such an instrumentation approach and we believe that the slowdown can be further reduced with the addition of sampling and optimizations.

A. Coverage

Before examining the accuracy of PSnAP for the benchmark set as a whole, it is important to ensure that individual instructions are being properly represented. This cannot be done unless a significant portion of instructions are represented by the defined stride patterns from section II-B. A coverage test was conducted to measure the number of instructions in the benchmarks that qualified as candidates. Of the over 30,000 instructions evaluated in the NPBs, only 29 are not represented using the defined patterns.

[4]http://www.sdsc.edu/us/resources/dash/dash_system_access.html

TABLE IV
Instruction-level pattern coverage statistics for NPBs.

Label	**BT**	**CG**	**EP**	**FT**	**IS**	**LU**	**MG**	**SP**
P	3455	384	30	231	62	2987	753	5262
R	0	2	0	0	11	0	16	0
T	3455	386	32	231	73	2987	769	5262

P=PSnAP, R=Random, T=Total

Table IV shows the coverage statistics for the categories in PSnAP. The random instructions, which occur in CG, IS and MG, were manually examined. Each of them results from the use of index arrays and are in fact not candidates for PSnAP representation. For these cases, a random access stream is generated during replay.

B. Pattern Resolution

The reproduction of fine-grained patterns has historically been a weakness in synthetic stream generation. A high level of resolution is very important when studying memory behavior, especially when prefetching is a consideration. In order to demonstrate the level of resolution achieved by PSnAP we manually compared the first 100,000 addresses for the dominant basic block in each benchmark. An exact match was achieved for each of the NPBs with qualifying instructions, even the complex pattern shown in Figure 1(a).

C. Cache Simulation Results

Cache simulation comparisons allow for a higher-level view of PSnAP's accuracy. We have shown that the fine-grained patterns are reproduced on a per instruction, and per basic block basis, but the lossy nature of the control flow compression leaves an opening for differences between the observed and synthesized streams. We compare the overall cache hit rates for the entire execution as well as perform a more fine-grained basic block comparison.

This evaluation uses a set of seven memory hierarchies taken from recent and historical HPC systems. Included in the set are PowerPC, IT2, MIPS, Opteron, Budapest, Nehalem, and IBM Power6. The MIPS structure is altered to create four hypothetical structures. The line size and associativity are varied in turn. The structures were chosen to represent small, medium and large caches, with a variety of line size.

The observed address stream of each benchmark was fed into a series of cache simulators [18], [14]. The cache simulations produce cache hit rates for each basic block in the application and for the entire execution. These cache hit rates are then compared with the cache hit rates that result from the simulation driven by the synthetic streams.

Table V shows the difference in the hit rates (between synthetic and observed) for the entire execution using the Nehalem cache structure. The accuracy varied very little across cache configurations and the Nehalem results are representative of the full set. PSnAP's overall accuracy is very high. The average error for L1 is 0.8%.

A more detailed comparison of basic block cache hit rates confirms PSnAP's accuracy. For this comparison the cache

TABLE V
PERFORMANCE OF PSNAP ON NPBS (4 PROCESSORS).

Benchmark	Full Trace Size	Compressed Files		% Abs Err in Cache Hit Rates (over full execution)		
	(GB)	ratio	size KB	(L1)	(L2)	(L3)
BT.A	1,120	84,856X	13,840	0.2	0.1	0.1
CG.A	18	83,020X	232	0.4	0.4	0.1
EP.A	51	1,973,790X	27	0.0	0.0	0.0
FT.A	64	97,203X	690	0.2	0.5	0.4
IS.A	43	134,019X	338	1.3	0.2	0.1
LU.A	599	79,399X	7,908	1.9	1.5	0.6
MG.A	40	19,760X	2,118	1.1	0.7	0.3
SP.A	508	33,359X	15,968	1.4	1.0	0.7

hit rates across all cache structures were compared for the dominant basic block in each benchmark. Table VI shows the observed and synthetic cache hit rates for the dominant basic block of each benchmark using the Nehalem cache structure. Not surprisingly, IS is the worst performing. The dominant instruction in IS performs a load calculated using an index array. It is categorized as undesignated and a random function is used to generate the synthetic stream. All of the basic blocks containing instructions with recorded stride patterns performed well, the maximum difference is 0.71%.

TABLE VI
ACCURACY OF PSNAP ON NPBS AT THE BASIC BLOCK LEVEL (4 PROCESSORS).

Benchmark	Cache Hit Rate		
	Observed	Synthetic	Diff.
BT.A	94.71	94.71	0.00
CG.A	75.43	76.14	0.71
EP.A	93.75	93.75	0.00
FT.A	86.09	86.48	0.29
IS.A	96.33	93.28	3.05
LU.A	93.77	93.73	0.04
MG.A	88.51	88.99	0.48
SP.A	97.70	97.70	0.00

Our experimental results demonstrate very clearly that the synthetic streams are very similar to the observed in terms of performance. The error is consistently below 1%, with the exception of IS.

D. Size and Slowdown

The size and scaling behavior of the memory profiles are major advantages of the PSnAP approach. Each of the benchmarks used for the accuracy evaluation produced memory profiles of less than 2MB, which is easily shared among collaborators. Table V shows the compression ratios for PSnAP. The size of each PSnAP profile is a function of address stream complexity rather than running time, or dataset size. Specifically, the size of the profile is a function of loop count, basic block count and instruction count, but is largely dominated by the memory instruction count.

Table VII shows that the collection overhead is on average 90X. This is very small for a profiling technique with no sampling. The addition of sampling is planned as future work, providing the opportunity for even more improvements.

TABLE VII
RUNNING TIMES FOR INSTRUMENTATION AND REPLAY.

Benchmark	Running Times(slowdown)		
	Exec. sec.	PSnAP sec.	Replay
BT.A	27	1856(69X)	1170
CG.A	0.7	56(77X)	42
EP.A	3	163(59X)	18
FT.A	2	197(116X)	235
IS.A	0.6	62(112X)	44
LU.A	19	1610(86X)	967
MG.A	0.8	88(117X)	109
SP.A	23	1441(62X)	1304

The replay time is a significant metric because slow replay leads to slow simulations. The current state of the art for replay speeds is to replay at disk read speed (optimistically 250 MB/s for a single disk without RAID). An optimized version of the PSnAP replay tool approaches this rate for some profiles. However, we have devised an alternative replay process in order to guarantee consistent replay speeds at disk speed.

We have demonstrate that the PSnAP profiles have a high degree of accuracy, low instrumentation costs, and that is is possible to replay the profiles at disk speeds. The result is a highly accurate, highly compressed trace that is easy to capture and replay whether to drive simulation or other analysis.

V. RELATED WORK

PSnAP is not the first attempt at taking advantage of regularities in address streams for compression. The following discussion addresses previous work done in the areas of address stream compression, synthetic address stream generation, and synthetic benchmark generation. PSnAP differentiates itself from all of the work described below because of the granularity of analysis, the compression ratios achieved, the overhead of collection and replay, and the fact that the profiles are human readable and lend themselves to manipulation.

Several schemes have been developed specifically for address streams and they are able to achieve a much higher level of compression than general compression schemes, up to six orders of magnitude (Sequitor [19], VPC [20], Mache

[21], and SIGMA [22]). These schemas have two main disadvantages. First, the time required to perform compression is very long and second the compression ratio is unpredictable because it depends on finding regular patterns in the address stream and treats all streams the same, even when the desired patterns do not exist. Each of these methods is lossless, meaning that even areas of random accesses are saved. Attempting to compress random accesses takes a large amount of time, and in most cases the compression ratio is insignificant. Sequitor is the standout performer of this group. It works by creating a context free grammar based on the patterns repeated in the address stream. The grammar is created dynamically during compression.

ScalaMemTrace [23] is a recently developed tool that works on the same principles as PSnAP. A hierarchical representation is used to represent the access patterns created by specific instructions. The representation uses and extended type of Regular Section Descriptor (RSD). Our method of collection has a smaller time complexity (PSnAP depends on the number of unique strides rather than the number of unique addresses) and the processed representation is slightly smaller. The methods are complimentary in that ScalaMemTrace is comparable with multi-threaded applications; a feature that PSnAP does not have.

A lossy compression technique was presented by Gao et al. [16] referred to as Path Grammar Guided Trace Compression (PGGTC). PGGTC works in a very similar fashion to Sequitor with the exception that rather than creating a context free grammar on-the-fly it uses static analysis to build a control flow graph, which can be used to create a context free grammar. Gao also realized that some portions of an address stream are truly random and therefore do not lend well to compression. Rather than attempting to compress them, they are detected, summarized and regenerated. This summarization is what makes this compression technique lossy.

Also directly related to our work are other efforts to summarize application behavior and generate synthetic traces. Sorenson et al. [24] expanded on work done by Grimsrud et al[25] that demonstrated that none of the five well-known approaches to this area achieved a high level of accuracy. The general categories examined are: the Independent Reference Mode (IRM), the Stack Model, the Partial Markov Reference Model, the Distance Model, and the Distance-Strings Model. Each of these was shown to not preserve the access patterns and locality demonstrated in the original trace. IRM came the closest to achieving this, but missed important features of the trace.

Even though other groups have attempted to classify program behavior based on the locality characteristics of their address streams [26], [27], [28], [29], [30], [31], [25], achieving within 90% verisimilitude when using these streams as representatives of the full application to predict cache hit rates has not been possible until a project called Chameleon[32]. Chameleon focused on the stream and application as a whole, this work breaks them down into their constituent pieces. That change in granularity allows for the extrapolation of address profiles that was not previously possible.

In addition, the first version of PSnAP [15] created synthetic address streams based on a statistical description of the strides encountered during execution.

A second type of synthetic trace generation is one that focuses on creating a full synthetic benchmark rather than the memory access patterns alone. Iyengar et al. [33] proposed a method of generating synthetic traces using a graph representation that characterizes each benchmarks program trace. The graph is a trimmed version of the control flow graph. The memory accesses are created by following the control flow graph using a statistical model and following a set of rules about how memory accesses within a given window relate. Joshi [34] proposed a technique for creating synthetic benchmarks based on the control flow graph as well. The memory locality was characterized by recording the most common stride and the frequency of that stride for each instruction. This is similar in concept PSnAP, but would miss complex patterns like the one observed in FT. Another project that aims to create synthetic benchmarks is [35]. In this work dynamic profiling is coming with static information to create synthetic code in a high level language that will have similar performance characteristics to the profiled code. PSnAP is different from this work in that the main focus is on storing profiles and creating traces for simulation. It is possible to use PSnAP to save instruction mix information as well and, therefore, theoretically create synthetic benchmarks.

VI. Conclusion

Despite the general usefulness of trace-driven memory simulation the collection, handling and storage of memory address traces remains problematic. PSnAP addresses the challenges associated address trace collection including space costs, time costs, accessibility, and proxy inaccuracies by decomposing address streams into smaller simple streams. The repeated patterns in the smaller streams are recorded during dynamic execution using a simple stride history scheme.

Space Costs. PSnAP achieves consistent compression ratios higher than any previously compression technique. Not only are the compression ratios high, but the growth of the profile sizes does not grow in relation to execution time. Once each instruction level stream description has been established the profile stops growing regardless of the number of times that it is repeated.

Time Costs. The compression ratios are achieve while maintaining low execution overhead (on average 90X). We have presented a method for replay that matches the disk speed available. It is also possible to perform direct replay of the profiles, which outperforms disk reads for some benchmarks. Another advantage of PSnAP is that specific areas of interest in the application can be captured in the profiles rather than the entire application. This is desirable for extremely long running applications where any slowdown is a difficult challenge.

Accessibility. PSnAP profiles are human readable and manipulatable. This opens up several new usage scenarios for address streams. Due to their size PSnAP profile are easily shared between collaborators. It is also possible to replay only specific portions of the stream. Specific loops can be identified and played along with the loops nested within them.

Replay check-pointing is also possible. This means that the replay can be performed in sections. It is possible to request only a million addresses at a time. This level of control makes exploring and experimenting with the streams much easier than with other compression methods.

Proxy Inaccuracies. It is not necessary to evaluate benchmarks or application kernels as proxies to real application. The applications themselves can be represented with PSnAP profiles.

PSnAP represents a real advance in the handling of address streams. We note improvements in the areas of space and time costs as well as making the information captured in memory address streams easily accessible to users.

PSnAP combines previous work done on dynamic control flow graph compression with a new technique for compressing instruction-level address streams. The development of the fine-grained instruction-level address stream compression allows for much faster instrumentation as well as the very small profiles. We demonstrate that fine-grained patterns are reproduced address for address and cache hit rates are reproduced with an average error of 0.8%.

References

[1] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai, "Challenges in computer architecture evaluation," *Computer*, vol. 36, no. 8, pp. 30–36, 2003.

[2] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, pp. 78 – 117, 1970.

[3] P. Calingaert, "System performance evaluation: survey and appraisal," *Commun. ACM*, vol. 10, no. 1, pp. 12–18, 1967.

[4] W. Anacker and C. P. Wang, "Evaluation of computing systems with memory hierarchies," *IEEE Transactions on Electronic Computers*, vol. EC-16, no. 6, pp. 670–679, December 1967.

[5] W. Anacker and C. Wang, "Performance evaluation of computing systems with memory hierarchies," *Electronic Computers, IEEE Transactions on*, vol. EC-16, no. 6, pp. 764–773, Dec. 1967.

[6] C. Olschanowsky, L. Carrington, M. Tikir, M. Laurenzano, T. S. Rosing, and A. Snavely, "Fine-grained energy consumption characterization and modeling," in *DOD High Performance Computing Modernization Program User Group Conference*, 2010.

[7] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead," 2008.

[8] X. Gao, "Reducing time and space costs of memory tracing," Ph.D. dissertation, University of California at San Diego, La Jolla, CA, USA, 2006.

[9] X. Gao, M. Laurenzano, B. Simon, and A. Snavely, "Reducing overheads for acquiring dynamic traces," in *International Symposium on Workload Characterization*, 2005.

[10] J. Flanagan, B. Nelson, and G. Thompson, "The inaccuracy of trace-driven simulation using incomplete multiprogramming trace data," in *MASCOTS*, 1996.

[11] D. R. Kaeli, "Issues in trace-driven simulation," in *Performance Evaluation of Computer and Communication Systems*. London, UK: Springer-Verlag, 1993, pp. 224–244.

[12] R. C. Murphy and P. M. Kogge, "On the memory access patterns of supercomputer applications: Benchmark selection and its implications," *IEEE Trans. Comput.*, vol. 56, no. 7, pp. 937–945, 2007.

[13] (2008). [Online]. Available: http://www.nas.nasa.gov/Resources/Software/npb.html

[14] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "Pebil: Efficient static binary instrumentation for linux," *International Symposium on the Performance Analysis of Systems and Software*, March 2010.

[15] C. Olschanowsky, M. Tikir, L. Carrington, and A. Snavely, "PSnAP: Accurate Synthetic Address Streams Through Memory Profiles," in *Workshops on Languages and Compilers for Parallel Computing*, 2009.

[16] X. Gao, A. Snavely, and L. Carter, "Path grammar guided trace compression and trace approximation," *International Symposium on High-Performance Distributed Computing*, vol. 0, pp. 57–68, 2006.

[17] R. E. Tarjan, "Edge-disjoint spanning trees and depth-first search," *Algorithmica*, vol. 6, no. 2, pp. 171–185, 1976.

[18] M. Tikir, M. Laurenzano, L. Carrington, and A. Snavely, "The pmac binary instrumentation library for powerpc," in *Workshop on Binary Instrumentation and Applications*, 2006.

[19] S. Mitarai, M. Hirao, T. Matsumoto, A. Shinohara, M. Takeda, and S. Arikawa, "Compressed pattern matching for SEQUITUR," in *Data Compression Conference*, 2001, pp. 469+.

[20] M. B. Computer, "Vpc3: A fast and effective trace-compression algorithm," in *IEEE/USP International Workshop on High Performance Computing*, 1994.

[21] A. Samples, "Mache: No-loss trace compaction," University of California at Berkeley, Tech. Rep., 1988.

[22] L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia, "Sigma: A simulator infrastructure to guide memory analysis," in *In Supercomputing*, 2002, pp. 1–13.

[23] S. Budanur, F. Mueller, and T. Gamblin, "Memory trace compression and replay for spmd systems using extended prsds?" *SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 30–36, March 2011. [Online]. Available: http://doi.acm.org/10.1145/1964218.1964224

[24] E. Sorenson and J. Flanagan, "Evaluating synthetic trace models using locality surfaces," *IEEE International Workshop on Workload Characterization*, pp. 23–33, Nov. 2002.

[25] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson, "On the accuracy of memory reference models," in *the international conference on Computer performance evaluation : modelling techniques and tools*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1994, pp. 369–388.

[26] R. B. Bunt and J. M. Murphy, "The measurement of locality and the behaviour of programs," *Comput. J.*, vol. 27, no. 3, pp. 238–253, 1984.

[27] D. Thiebaut, J. Wolf, and H. Stone, "Synthetic traces for trace-driven simulation of cache memories," *IEEE Transactions on Computers*, vol. 41, no. 4, pp. 388–410, 1992.

[28] P. J. Denning and S. C. Schwartz, "Properties of the working-set model," *Commun. ACM*, vol. 15, no. 3, pp. 191–198, 1972.

[29] K. Beyls and E. H. D'Hollander, "Reuse distance as a metric for cache behavior," in *In Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, 2001, pp. 617–662.

[30] W. S. Wong and R. J. T. Morris, "Benchmark synthesis using the lru cache hit function," *IEEE Trans. Comput.*, vol. 37, pp. 637–645, June 1988. [Online]. Available: http://dl.acm.org/citation.cfm?id=45868.45869

[31] A. Agarwal, J. Hennessy, and M. Horowitz, "An analytical cache model," *ACM Trans. Comput. Syst.*, vol. 7, no. 2, pp. 184–215, 1989.

[32] J. Weinberg and A. Snavely, "Chameleon: A framework for observing, understanding, and imitating memory behavior," in *Workshop on State-of-the-Art in Scientific and Parallel Computing*, Trondheim, Norway, May 2008.

[33] V. S. Iyengar, L. H. Trevillyan, and P. Bose, "Representative traces for processor models with infinite cache," in *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, ser. HPCA '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 62–. [Online]. Available: http://dl.acm.org/citation.cfm?id=525424.822668

[34] A. Joshi, L. Eeckhout, R. H. Bell, Jr., and L. K. John, "Distilling the essence of proprietary workloads into miniature benchmarks," *ACM Trans. Archit. Code Optim.*, vol. 5, pp. 10:1–10:33, September 2008. [Online]. Available: http://doi.acm.org/10.1145/1400112.1400115

[35] L. Van Ertvelde and L. Eeckhout, "Benchmark synthesis for architecture and compiler exploration," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, dec. 2010, pp. 1 –11.