Path Grammar Guided Trace Compression and Trace Approximation

Xiaofeng Gao
xgao@cs.ucsd.edu

Allan Snavely
allans@sdsc.edu

Larry Carter
carter@cs.ucsd.edu

Abstract

Trace-driven simulation is an important technique used in the evaluation of computer architecture innovations. However using it for studying parallel computers and applications is at best very challenging. Acquiring, representing and storing the traces are among the major issues. In this paper, we introduce path grammar guided trace compression (PGGTC) and effective address trace approximation (TA) to speedup compression and reduce trace sizes. PGGTC relies on static analysis to build rules and determine actions to guide online trace compression. Combined with gzip, PGGTC can compresses control flow traces over 330 times smaller than using gzip alone. Compared to the widely popular Sequitur algorithm alone, PGGTC with gzip is on average 40 times faster, while the traces are only 3 times bigger. PGGTC can be also used with Sequitur to double the compression ratios of Sequitur by itself and do it 14 times faster than Sequitur by itself.

Address traces of parallel applications with significant randomness are often impossibly large even after being compressed with any loss-less scheme including PGGTC. For effective address trace reduction, we introduce Trace Approximation (TA). Performance-wise similar effective addresses are generated based on very compact summaries of how the memory is accessed during each structure instance instead of compressing them. We demonstrate two approaches: Selective Dumping and Memory Signatures, to summarize the properties of effective address sequences. Both approaches are validated by feeding the generated approximate trace to cache simulators of 25 different configurations. The simulated results are very close to the simulation results based on full effective traces while the selective dumped address or memory signatures require several order of magnitude less disk space to store. In summary, we move trace-driven simulation into the realm of the feasible for larger parallel machines and applications.

1 Introduction

Trace-driven simulation is an important technique used in the evaluation of computer architecture innovations. In fact 80% to 90% percent of research papers proposing new architecture features over the recent years have used trace-driven simulation [1, 2] to validate their benefits. In this paper we address some difficulties in acquiring, representing, storing, and replaying event traces of programs for the purpose of such trace-driven simulation.

The most common way of obtaining event traces is to instrument the program's source code or binary and then execute the resulting instrumented version of the program, thereby to capture and record events and associated information. This process is notoriously slow. The slowness is so sever that many times studies have to be limited to small kernels or benchmarks due to the length of time required to obtain traces of anything more meaningful. Our previous work [3, 4] has focused on making this process faster. Unfortunately, even when this process is faster, the resulting traces are huge and unwieldy. Just for example, a full uncompressed trace of the effective addresses touched by the NAS Parallel Benchmark [5] BT Class A run on 4 processors takes more than one quarter of a terabyte to store. It should be clear the average user would run out of disk space for acquiring memory traces of full-scale parallel applications. Various compression schemes have been proposed to reduce disk storage requirements. These will be further examined below but they all have a fundamental limitation; these compression methods may or may not result in much compression; it depends on the patterns of events in the raw trace and the compression scheme's ability to recognize and summarize those patterns. A very common and fast compression scheme, gzip [6, 7], is based on variable-to-fixed length coding. Many reasons can make it less efficient in compressing traces, especially address traces. Methods that are more sophisticated are notoriously slow. For example, Sequitur [8], a widely used compression scheme for architectural research, takes 20 hours (!) to compress just the control flow trace of BT Class A.

To make event traces storable in reasonable-sized files without requiring potentially huge (unknowable in advance)

amounts of scratch disk, we introduce PGGTC, Path Grammar Guided Trace Compression. This technique keeps control flow information around during the dynamic step and uses it to compress event traces on-the-fly. We call such compressed control flow trace files Structured Path Histories (SPHs) . Combined with gzip, PGGTC on average is over 40 times faster than methods such as Sequitur and requires no scratch disk. Its compression ratios are not quite as high as Sequitur but much better than directly using gzip and can do better than Sequitur after post-analysis, thus hitting a sweet spot for time and space required to capture and store event traces. Most importantly, because control flow information is preserved and stored with the event trace as part of the compression scheme, one can flexibly decompress only events pertaining to code sections of interest. This flexibility makes dealing with large event traces more wieldy; it also enables some advanced uses such as synthetically generating an event trace for the program run on an input different from that with which the original trace was gathered.

We also deal with the following problem: some kinds of program events defeat even sophisticated compression. A good example is a sequence of truly random effective memory accesses. If there is no detectable pattern to these addresses there can be, by definition, no summary to fully characterize them. Such events are the bane of the tracer as they cost an inordinate amount of time to gather and cannot be compressed. By contrast, highly regular event patterns can often be determined via low-overhead static analysis before tracing and then just a few details, such as number of times the pattern is encountered in branch/loop trip counts etc. can be filled in by lightweight tracing. But potentially non-patterned events, such as the aforementioned random memory addresses that might result from a series of pointer references for example, need to be monitored dynamically during tracing and may result in sequences that are hard or impossible to compress. We approach this problem with a variety of techniques. We apply static analysis extensively and focus on dynamic tracing on only those events that cannot be determined by static analysis 2) We use sampling to statistically characterize potentially random events without storing all of them. 3) We extend this idea to Trace Approximation (TA).

TA leverages the following idea: many times, for trace-driven simulation and validation of benefit of architectural features, one would be satisfied with a 'performance similar' trace. That is, from the performance standpoint, it may be that any sequence of random memory accesses with the same statistical distribution would perform the same against the architectural feature being evaluated. Therefore one can replace traces of random addresses with other (more compressable) random addresses. Using our TA technique one can obtain a compact event trace from a program that, when replayed against a simulator, will reproduce the statistical behavior of the events (statistics defined by the user) of the original program without guaranteeing exact duplication of particular events in sequence (such as exact sequence of random memory addresses touched).

We show PGGTC, SPH and TA are practically very useful, enabling the gathering and recording of event traces in tractable time and using tractable disk space; SPH traces are exact. We validate the TA traces against a set of cache simulators, showing the results have high verisimilitude compared to full traces.

Section 2 evaluates limitations of current trace compression schemes. Section 3 introduces PGGTC and the properties of the SPHs. Section 4 treats effective address tracing as a special case of tracing events that may have considerable randomness in them and introduces TA. Section 4.4 validates the approximate effective addresses. Section 5 is related work.

2 Limitations of Current Compression Schemes

Trace files can be compressed to reduce their storage size. The most common way is to just use gzip. gzip commonly results in about two order-of-magnitude reduction in file size. There have been many techniques developed to compress control flow traces and associated effective address traces to achieve, commonly, as high as 6 orders of magnitude reduction in file size. These include Sequitur, VPC [9] and SIGMA [10]. However they are much slower than gzip. They also can achieve high compression ratios only when there are detectable patterns in the address sequence. Uncertainty as to compression ratio is very undesirable because, for a previously unstudied application, the user has no clue of disk resource requirements for saving the trace or even feasibility of saving the trace. One might need a few kilobytes of storage or several terabytes.

Let us examine some scenarios where Sequitur does poorly. The Sequitur algorithm tries to build a context free grammar from a sequence (for example of blockids) and uses this grammar to compress the trace. The algorithm goes through the elements in the sequence and creates rules based on observed repetitions. If there is a sub-sequence that has been repeated more than a threshold, the algorithm creates a reduction rule and replaces future appearances of the sub-sequence with the rule. The process repeats recursively until there are no more rules to create. For example consider an inner loop shown in Figure 1. Suppose in block C, function 2 is called and it always returns to the caller. For the following trace:

```
XACFGGHHIBDABDACFGHIBDABDACFGGGGHHHI
BDABDACFGHHHIBDABDE
```

the Sequitur algorithm has created the following rules:

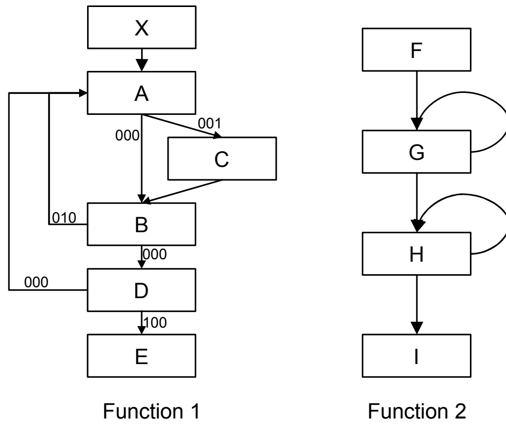

Figure 1. Control Flow Example

```
0 -> X A 1 2 3 3 G G 2 H 3 H H 4 5 E \n
1 -> C F G
2 -> G H
3 -> 4 6 1
4 -> H I 6
5 -> B D
6 -> 5 A
```

The trace can be recovered by replacing non-terminals (numeric rules) in rule 0. Examination shows that most of the rules Sequitur has "discovered" are just recaps of what is obvious in the control flow graph.

Further examination highlights limitations of Sequitur and others stream compression techniques, which stem from the fact they treat program traces as streams of completely unknown events without regard to the control flow. In addition, the hierarchical nature of the program trace (i.e. nested loops and function calls) is not considered by algorithms such as Sequitur. The detected patterns may be mixed up between functions and loops and branches etc. Many patterns indeed will not be detected because they are obscured by some non-patterned elements that happens in the middle from a function call and the like. In the example above Sequitur fails to reveal a much more interesting pattern in the loop: two paths are taken alternatively ACBD, ABD, ACBD, ABD.

These algorithms generally are troubled by non-patterned elements in the sequence. Too many rules and patterns are created to attempt to summarize things that may not have a pattern at all. These rules and patterns increase the sizes of look up tables and make the compression process much slower than it would otherwise be. For example, as will be seen in the results section, Sequitur has problems with NAS Parallel EP, which has a major loop which is basically random. Sequitur could not finish compressing the 944MB control flow trace in 70 hours.

3 Path Grammar Guided Trace Compression

3.1 Path Grammar and Actions

Building a context free grammar that basically captures (at least some aspects of) the CFG as Sequitur does, is redundant for the purposes of compressing an application trace. In fact the control flow is determined by the static control flow graph, which is already a grammar and is already known at compile time and can be used directly for compression. We first parse the binary to build a CFG by identifying basic blocks, edges and natural loops. Then basic blocks are assigned to structures. A structure is either a natural loop, or a function. Each basic block (single entry, single exit instruction sequence) is only assigned to the innermost structure that contains this block. Each structure has its own path grammar; the grammar is built from the sub-graph that only contains the basic blocks that belong to this structure. These grammars will be used to encode paths taken within the structure during execution;

A path value represents a particular path through a structure and is encoded as a bit string. For each branch in a structure, a unique bit is reserved. Taking one edge will set the bit to zero and the other edge will set the bit to 1. This is a similar approach to that used in DYNAMO [11] and TFP [12] but we use one bit per branch instead of one bit per block to reduce the number of bits required. Now for most cases, comparing two path values is effectively comparing two unsigned 32-bit integers as most structures do not contain more than 32 branches. For all the NPB benchmarks, only 14 structures required bit string longer than 32 bits. For the rare structures that require more than 32 bits to present all its branches, the bit string is hashed into a 32-bit value that must be looked-up during tracing and compression. In the inner loop in Figure 1, there are three branches so three bits are required for representing the paths in this structure. For example the first branch starting from block A is associated with the lowest bit. When edge (A,C) is taken, the bit will be set 1 (the path value would be xx1 depending on the other two branches).

For tracing and compression, edges are are associated with actions based on the grammar, such as to set the bit strings. Unlike other compression schemes, these actions are determined during *static* analysis and blocks are instrumented accordingly so there is no need to create rules and look up a dictionary to determine what should be done when a particular edge is taken. For example the actions assigned to edge (D,A) include the following three actions. First the current path value is BitOred with its bit mask (000). Secondly (D,A) indicates a completion of current path. The path value is added to a history buffer. Thirdly the path value is reset for the next pass of the loop. The structures of the control flow graph shown in Figure 1 are defined as

```
struct0={X,E}
struct1={A,C,B,D}
struct2={G}
struct3={H}
struct4={F,I}
```

Natural loops sharing the same loop head are regarded as one structure. For structure 1, static analysis will determine the bit mask for each branch and the actions for each edge. Not all edges need actions. Actions that BitOr 0 can be omitted. Here now are the actions associated with the edges of structure 1:

```
(X,A)=>{PushStack()
        struct1.NewInstance()}
(A,C)=>{struct1.pathval.BitOr(001)
        RegisterCallSite()}
(B,A)=>{struct1.pathval.BitOr(010),
        struct1.AddPathValToHistory(),
        struct1.ResetPathVal()}
(D,A)=>{struct1.AddPathValToHistory(),
        struct1.ResetPathVal()}
(D,E)=>{struct1.pathval.BitOr(100),
        struct1.AddPathValToHisotry(),
        struct1.CompleteInstance(),
        PopStack()}
```

3.2 Structured Path History

PGGTC compress the traces for each structure individually and stores them independently. The compressed control flow trace files are called Structured Path Histories (SPHs). Although SPH also deals with whole program path histories, it differs from Whole Program Path (WPP) [13] in three major aspects. First the path histories are recognized, processed and saved individually and differently based on structures. Post-processing can work on each structure's history without decompressing the whole trace. It is not only easier and faster to work on parts of a trace like this, patterns are easier to identify when elements from other structures are not intermixed. Secondly the paths and actions are defined during static analysis. The online analysis is considerably lightweight compared to other approaches which have to build the dictionary during the online analysis. Thirdly the grammar is the *same* for traces from different runs of the same application.

Once the structures are defined and actions have been associated with edges, blocks are instrumented accordingly to implement the defined actions. Online analysis will capture the blockids, use one look back entry to determine the edges and follow the actions associated with the edges to compress the trace. When an edge completes a path through a structure, for example either a back edge such as (D,A) or an exit edge such as (D,E), then the bit string representing the completed path value is added to a history buffer attached to that structure. A structure's history buffer is organized as a pair of values consisting of a path value (bit string) and a repetition integer. If the bit string is the same as the previous one in the buffer, the repetition integer is incremented, otherwise a new pair of values is added to the buffer. When a loop instance completes, a special mark is added to the buffer to indicate the completion of that instance of the loop so that future instances of the loop will start from a new pair. When the buffer is full, we simply use any loss-less compression scheme, such as the utility functions provided in zlib [14] (gzip) to compress it and append it to the structure's SPH. Such a file is then a history of the structure recording all the paths taken in every dynamic instance of it.

Recursion must be dealt with. An execution stack is used in our implementation to overcome the problems caused by multiple instances of the same structure. Every time a path is completed, it is added to the history of the most recent instance of the structure. Keeping the execution stack increases overhead significantly, if recursion is guaranteed not to occur, the implementation could be sped up by using a simple version. The trace shown previously in Section 2 is represented by the following path histories of 5 structures. As an example, (0:3,1:1.) indicates Path 0 is taken three times and then followed by Path 1 once and the loop instance completes.

```
struct0:{(0:1)}
struct1:{(1:1,0:1,1:1,0:1,1:1,0:1,1:1,
          4:1.)}
struct2:{(0:1,1:1.)(1:1.)(0:3,1:1.)
         (0:1.)}
struct3:{(0:1,1:1.)(1:1.)(0:2,1:1.)
         (0:2,1:1.)}
struct4:{(0:4)}
```

The targets of certain instructions are unknown during static analysis. They can also have more than 2 followers during execution. Such instructions include jump and call instructions using registers for addressing. Blocks ends with such instructions are called *call sites*. The online analysis keeps a buffer of followers for each of these call sites. Normally a call site has only one or a very limited number of followers, so gzip can compress the follower buffer with high compression rate.

When one wants to examine or replay the trace, path values are replaced with sequences of blocks. For example, the decompression process will replace path0 of structure 1 with ABDA. When a block in the sequence is a call site (for example block C) , its next follower and the follower's instance are filled in after the call site. Each structure maintains its own pointer to the next instance from its history file. Here are some path values decompressed to sequences of blocks for structure 1:

```
path0(val=000) => ABDA
path1(val=001) => AC(...)BDA
path2(val=010) => ABA
path3(val=011) => AC(...)BA
path4(val=100) => ABDE
path5(val=101) => AC(...)BDE
```

SPH is a loss-less representation of the control flow trace. The correctness can be validated by comparing the original stream of blockids to the decompressed ones. PGGTC achieves high compression ratios for its SPHs by transforming a sequence of events into a representation that is friendly to low-level compression schemes. It does this by keeping separate histories for each structure and not allowing them to get intermixed. Because each structure has its own bit string, the possible values of the bit string depends on how many decision points (branches) there are in the structure. For the most important inner loops of scientific applications, there are not too many branches. They may be typical for example of G and H in Figure 1 which have only have two possible path values. A limited range of path values enables gzip to achieve very high compression ratio as will be seen in the results (next).

3.3 Results

The NAS Parallel Benchmarks were instrumented, and their control flow traced, using ATOM [15] on an Alpha-based supercomputer at the Pittsburgh Supercomputing Center. Each basic block in each application was instrumented to write out the blockid. Although it is not necessary to save the trace first onto disk with PGGTC, we did this in the following set of experiments just to compare the speed of PGGTC compression against other methods. The compression experiments were all done on an Intel workstation with 1GB memory and a 2.8GHz P4 processor. The results are shown in Table 1.

PGGTC+gzip, which uses gzip to compress path histories, is on average 24 times slower than using gzip, but the sizes of compressed files are on average over 300 times smaller. Sequitur source code was downloaded from `http://sequitur.info/` and applied to the uncompressed traces. Sometimes Sequitur yields compressed files as much as five times smaller than PGGTC with gzip, sometimes the compressed files are actually slightly larger, but the compression takes on average 40 times longer than PGGTC+gzip; this does not seem like a reasonable time/space tradeoff. In addition, Sequitur requires full trace on the scratch disk before compression and apparently has trouble handling big traces with highly random contents (such as EP trace).

Lastly, PGGTC was used with Sequitur as the underlying low-level algorithm for compressing path histories (PGGTC+ Sequitur). Because Sequitur source code doesn't support accumulative compression as gzip does, it is applied on decompressed SPH files from PGGTC+gzip. The time in the last column includes the extra time for PGGTC+gzip. PGGTC + Sequitur compresses traces to be more than twice smaller, and does it 14 times faster, than using Sequitur alone. The last row lists the average gain in compression ratio over gzip and the slowdown over using gzip.

3.4 Properties of SPH

The inherent program's hierarchical structure preserved in SPH, besides enabling good compression, enables study of the dynamic behaviors of applications. Unlike the other path based traces such as WPP, which generates one, possibly huge, monolithic file which is difficult to analyze and unwieldy, SPH breaks the trace according to its natural structures and stores the histories of each structure individually. Each file can be decompressed and analyzed independently. And these smaller files are much easier to handle. In addition, SPH encapsulates loop instance markers in the compressed traces. This enables one to perform extra analysis on higher levels such as detecting the repetitive patterns at the instance level. Currently, in order to reduce time overhead, PGGTC only detects immediate repetitive patterns at path level during online analysis. Post analysis can be used however to detect higher level patterns. For example, a repeated pattern in loop instances such as (1:1,0:1,1:1,0:1,1:1,0:1,1:1,4:1.) can be detected during post analysis. In addition, SPH enables one to detect the differences between two different runs of the same application, either for performance modeling or debugging. One major problem of diffing two traces is how to determine the synchronization points. With SPH, the traces can be compared at the structural level. It is easy then to tell which structure's behaviors are exactly the same and which are different during different runs (as with different inputs for example). The instance marker in the history can be used to align the comparison at the instance level. It is possible to pinpoint the exact instance at which two traces become different. Rules to encode paths are derived from static binary, so bit strings can be compared directly from different traces of the same application.

As an example of this diffing capability, we tested the code shown in Figure 2 with four sets of inputs where an input is a pair of loopcount and runlen: (18, 3) (20, 4) (33, 7) (45,5). A script is used to compare the traces. The script calls the UNIX diff command to compare the structure histories with the same file names in different directories. Only the files that are different are printed. In this case, the history file of structure 21 (the inner loop of AlterPath()) are different when different inputs are provided.

```
diffing  18_3/hist.21  20_4/hist.21
<7:1,5:3,7:1,5:3,7:1,5:3,7:1,5:3,7:1,1:1.
---
```

Application	uncompressed	gzip	PGGTC+ gzip	sequitur	PGGTC + sequitur
BT.A.4	3.26GB	15.5MB (77.9sec)	23.9KB (1384 sec)	14.0KB (19.9hrs)	5.81KB (2936sec)
CG.A.4	253.3MB	6.03MB (12.9sec)	2.39MB (156 sec)	349.2KB (1.5hrs)	241.2KB (551sec)
EP.A.4	944.5MB	15.2MB (30.7sec)	9.1MB (566 sec)	N/A(>70hrs)	N/A(>70hrs)
FT.A.4	558.9MB	3.6MB (14.0sec)	24.4KB (322 sec)	4.4KB (3.7hrs)	2.82KB (1007sec)
IS.A.4	343.8MB	506.1KB (6.9sec)	1.9KB (399 sec)	2.2KB (1.8hrs)	805B (1021sec)
LU.A.4	873.8MB	5.2MB (18.9sec)	17.1KB (466 sec)	6.6KB (5.4hrs)	4.25KB (1910 sec)
MG.A.4	431.8MB	14.9MB (10.2sec)	21.8KB (192 sec)	34.3KB (2.5hrs)	10.62KB (353 sec)
SP.A.4	5.04GB	26.9MB (121.6sec)	49.6KB (2616 sec)	28.5KB (31.5hrs)	8.27KB (3.3hrs)
Average		1X(1X)	331.6X(24.3X)	633.9X(868.1X)	1530.5X(76.3X)

Table 1. Compression Ratio of Control Flow Trace and Compressing Time

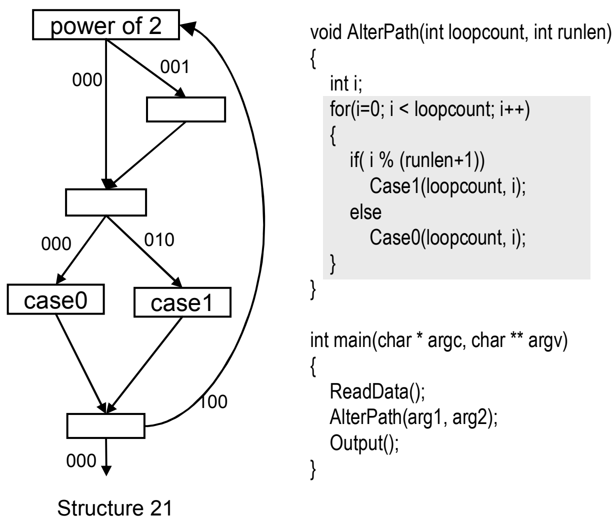

Figure 2. Alternating Control Flow

```
>6:1,4:4,6:1,4:4,6:1,4:4,6:1,4:3,0:1.
diffing  18_3/hist.21  33_7/hist.21
<7:1,5:3,7:1,5:3,7:1,5:3,7:1,5:3,7:1,1:1.
---
>7:1,5:7,7:1,5:7,7:1,5:7,7:1,5:7,3:1.
diffing  20_4/hist.21  45_5/hist.21
<6:1,4:4,6:1,4:4,6:1,4:4,6:1,4:3,0:1.
---
>6:1,4:5,6:1,4:5,6:1,4:5,6:1,4:5,6:1,4:5,
 6:1,4:5,6:1,4:5,6:1,4:1,0:1.
```

The high level point is that the compression scheme enables tractable comparison of traces at the structural level and can be used for simple, intuitive, comparisons of traces such as diffing.

3.5 SPH Alternation and Synthesis

If the kernel of an important applications is not stable and behavior is dramatically different depending on input, multiple traces should be used to feed the simulator so that the simulated results will be more representative of a real workload. Collecting, compressing, saving and replaying multiple traces of every possible input, even using PGGTC, be too time-consuming to be practical. To address this, SPHs can be artificially generated to reflect what would be the resulting traces from the program run with different inputs, and plug into existing SPHs collected from a real run. For example, the path histories of the inner loop of function AlterPath() in Figure 2 is not stable; histories are determined by the inputs. Compiler optimization inserted an extra branch in the loop. If the second parameter is 2^k-1, the first branch will always be taken so the possible path values are 7 (111) and 5 (101) for paths ending with the back-edge, and 3 (011) and 1 (001) for the paths that exit the loop. Otherwise the possible path values are 6(110), 4(100), 2 (010) and 0(000). Path 0 and 2 (or 1 and 3) can only be the last path because they both indicate the back-edge is not taken so they will complete the loop instance. The repetition of path 4 (or 5) will equal the second parameter. The length of the trace is determined by the first parameter. If such input-dependent behaviors are already known, one can simply generate the trace for just one set of parameters, say (45, 8) with the resulting SPH:

```
6:1,4:8,6:1,4:8,6:1,4:8,6:1,4:8,6:1,
4:7,0:1.
```

and then the trace for another set of parameters, say (108,15), will trivially be:

```
7:1,5:15,7:1,5:15,7:1,5:15,7:1,5:15,
7:1,5:15,7:1,5:15,7:1,5:10,1:1.
```

Synthetic SPH files can be used together seamlessly with SPHs of other structures that are acquired from actual instrumented runs. A generator can be used to systematically

generate SPHs to cover all the interesting cases without running the instrumented binary to collect them. Another extreme example is if it is known beforehand that paths of certain structure will be random (as is the case for EP), there is no need to waste time and space to collect and compress the traces from such a structure. The decompressor of that particular structure can be modified to issue random valid path histories instead of regenerating block sequences from recorded histories.

4 Address Trace Approximation

So far in the discussion we have focused just on control flow traces. Control flow traces are relatively small to begin with; address traces are much bigger and more difficult to compress. Yet address traces, that is the history of memory locations touched by an application, are among the most important event sequences for simulating how a program interacts with a machine's memory hierarchy [16]. Unlike control flow traces, repetition in effective addresses is less likely so repetition based compression schemes do not work well directly. Effective addresses are typically compressed using offsets, similar to Mache [17] and PDATS [18]. However even the most sophisticated compressing schemes do poorly when the address sequence is mostly random and resulting compressed trace is almost the same size as the uncompressed one. Such sequences are unfortunately common in scientific applications that deal with sparse data, so it is impractical to have any lossless compressed effective address traces for these applications. However, for many performance related studies that consume addresses traces, such as cache simulation, there is no need to keep the exact addresses in order to get the same or very similar simulation results. An address sequence approximately similar to the application's true address sequence, a sequence that can be represented in a compact manner, may be just as useful as long as it behaves in a manner similar to the original. In this section we discuss two approaches to collect information from traces for generating approximate ones 1) selective dumping and 2) summarized memory signatures. The SPHs are used as the backbone of the execution and approximated addresses are generated by "decorating" SPHs with additional address information. To validate accuracy we compare cache simulations against full address sequences and selective or approximated sequences.

4.1 Static Data-flow Analysis

For memory tracing our first strategy is to avoid it as much as possible. Dynamic memory tracing is notoriously slow. We attempt to ascertain as much as we can about the program's address sequence via static analysis (much faster) and identify the minimum set of load and store instructions in the binary that must be instrumented and traced to obtain a complete address trace. We leverage the fact many loads and stores have some constant static relationship and one needs only know one of the set to determine the others. In our previous work [3, 4], we showed that if every

App.	Inst.Level	Block Level	Mem Inst.
BT.A.4	300	1105	18714
CG.A.4	141	579	3489
EP.A.4	45	184	828
FT.A.4	225	633	4158
IS.A.4	22	155	462
LU.A.4	249	966	18396
MG.A.4	439	1124	9158
SP.A.4	338	1732	34166

Table 2. Total number of instrument points after and before static analysis

memory instructions is instrumented, then typically 10-fold slowdown over un-instrumented execution is a lower bound. The slowdown is typically much higher if there is any actual dynamic analysis. Such high overhead makes it infeasible to completely trace larger applications online. Fortunately, static data-flow analysis can be used to group memory instructions into static sets if the register values used by the instructions are guaranteed to be the same or to have some fixed, predetermined offset. Only one memory instruction is needed to be instrumented from each group, and the effective addresses of the others can be calculated based on the known offsets. On average, over 90% of the load/store candidates for instrumentation can be eliminated in this way. For the two schemes described in the next two sections, online tracing is done only on the loads and stores determined to be in the minimal set required to determine all address values. Table 2 shows the total number of instrumentation points required to ascertain all memory addresses compared to the total number of memory (load and store instructions) for the NPB. Our instrumentation uses the sum of the first two columns where the first column is the total number of load and store instructions instrumented for address tracing, the second is the number of instrumentation points required to do control flow tracing, and the last column is the total number of memory instructions in the program (candidates for memory tracing by a naive scheme).

4.2 Selective Dumping

Groups are then analyzed to determine static stride patterns. Loads and stores without detectable static strides can be sampled to understand how they access the memory. Sampling has already been shown to be effective in that processing only a small fraction of the overall addresses

and interpolating the others can generate simulation results with a high degree of verisimilitude to full traces [19, 20]. Our contribution is to focus sampling on just those instructions with no detectable static stride patterns. For example in the CG kernel, two of the three loads have static stride patterns. If the addresses are sampled equally, only one third of collected addresses are actually necessary. The effective addresses of the other two groups can be exactly regenerated given just their starting address and total number of accesses. Thus selective sampling is only applied to instructions without static patterns (as determined by static analysis). Given the same amount of disk space, Selective Dumping can provide much more address information.

SPH is used as the backbone to connect all the individual pieces and provide the sequential information required to regenerate the address sequence. In the simulation phase, simulators are driven by generated approximate addresses. The number of memory instructions and their order, offset, groups on a path are all statically known. Effective addresses are generated and issued according to the execution paths recorded in SPH. If a memory instruction belongs to a group whose static stride is known, the effective address is generated sequentially from the start address of the group(via tracing). For the memory instructions whose groups have been selectively dumped, the effective addresses are pulled from the sampled addresses.

4.3 Generating Effective Addresses from Memory Signatures

Many scientific applications are programmed in a refining style, which means the program changes behaviors over time. Selective dumping without awareness of phase changes can cause important data points to be missed. PGGTC can detect certain phase changes in control flow and these can be used to enable even more intelligent selective dumping. However like any other control flow based phase mechanism, the detected phases in control flow are not always correlated to actual changes in the address sequence and vice versa. For example for a loop that uses an index array to access a sparse matrix, the data access patterns can be completely different for different instances of the loop even if the control flow of two instance of the loop are exactly the same. This happens when the index array are assigned to different regions in the matrix which have different locality characteristics.

As we mentioned before, it may not be necessary to record the exact addresses for non-patterned groups. Our previous work [21] has shown that even uniformly random addresses generated in the observed range can be used to approximate addresses with fair success. PGGTC can be used to guide how and when memory access signatures could be summarized and stored according to more sophisticated statistical analysis. A memory access signature of a group is a summary of how the effective addresses of the instruction group are distributed in memory during one instance of the structure As with selective dumping, we only process addresses of groups without static stride patterns. Different analysis or simulators may have different focus on the properties of the address stream. For analysis that determines hot spots in memory, a histogram analysis of the addresses will be sufficient. We focus on cache simulation and in what follows the memory signatureis a tuple of 11 values. Two values record the region in memory space that are active during the instance of the structure. Eight values are used to record stride distribution indicate how the addresses jump within the region. Every captured address is compared to the previous address of the same group to calculate stride. This stride is then put into one of 8 buckets of strides, covering various stride values from less than 8 bytes, to over 512 bytes. The last value is used to measure the locality score in the short term cache. This short term locality cache is used to identify the spatial locality and reuse among addresses from all groups. A traditional way of finding short term locality is to have a global observation window which holds some previously captured addresses. Each captured address, from any group, is looked up in the window to determine the shortest distance to a previous address. To make this approach effective, the observation window has to be relatively big. Increasing the window size will substantially increase the overhead of on-line processing. To approximate this more quickly, we use a direct mapped cache to detect localities. Every address captured is used to update the cache, but the miss is only counted if the stride is larger than a threshold, for example 64 bytes. Unlike filter cache used in various filtering based trace reduction schemes, the cache here is not for discarding addresses. It simply is used to measure locality attributes.

The disk space required to save the memory signature is relative small and can be estimated from SPHs. For a group that is statically classified as strided, one value is required for each structure's instance. For other group, 11 extra values are needed per instance of the structure.

Approximate effective addresses can be re-generated from the recorded information as follows: If the group has been statically determined as strided , the addresses from this group are simply generated from the recorded (by tracing) start addresses and using the fixed stride. For such groups, the address sequence generated will be exactly the same as the the original. For the groups that are classified as non-patterned from static analysis, the addresses are generated from the signatures and could be different from the original. If all strides of the group are in the same bucket, the stride is the average stride calculated from lowest address, highest address and the counter of the bucket. The approximated address sequence starts from the lowest address and

is incremented by the calculated average stride.

For other groups, a random number [22] is used to select the stride based on the histogram of the stride bins. If the chosen stride is larger than the threshold, another random number is used to determine whether this access should be a hit or miss according to the miss rate in the locality cache. If this address is selected to be a hit, the address generation routine will check the cache and make sure the generated address is a hit otherwise it will be regenerated. It is worthwhile to note that although the cache is used only for strides larger than the threshold, all generated addresses, even from groups that are statically determined as strided, update the cache.

4.4 Results

4.4.1 Disk Requirement for Trace Approximation

SPHs are required to generate the approximate addresses from selectively dumped addresses or memory signatures. SPH can replay the whole execution from the very beginning or from any given entry point via an extra synchronization file. These synchronization files tell how the instances of nested structures are related. These files can be created from the SPH during post processing or created at the same time as the SPHs are being generated. For selective dumping we only dumped the addresses of selected structure's groups for the first 5000 dynamic structure instances. Such a small sampling period is sufficient in this case because the NPB benchmarks are inherently repetitive and have no dramatic phase changes. For real applications, longer and more frequent dumping for the selected structures may be necessary.

Table 3 shows the sizes of a full address trace of each of the NPB in column one, the size of the SPH information in the column labeled Size of SPH (column three), the size of the selective dump, memory signature, and aforementioned synchronization information required to replay the trace from any arbitrary starting point in columns four, five, and six respectively. It can be seen that selective dumps and memory signatures are very small compared to full address traces.

4.4.2 Validation of Approximate Traces

Table 4 compares simulation results using generated approximate trace from selectively dumped addresses and memory signatures to simulation results using actual traced addresses. The simulator simulates 25 cache configurations list in Table 5. It generates cache hit rates for each basic block and for each configuration. All blocks which have over 5000 total memory accessed are counted in the comparison. The values listed are the average absolute differences in hit rates of all counted blocks between the simulators driven by selectively dumped trace vs. simulators driven by full memory trace. It will be seen that errors are very low for either method of Trace Approximation on these benchmarks against these caches.

5 Related Work

PGGTC and the resulting SPH are closely related to various research related to whole program path (WPP) [13, 23]. The difference is already mentioned in Section 3.2. PGGTC is not a standalone compression scheme like LZW [7], Sequitur [8, 24] and VPC [9]. It parses the application traces and guides the lower-end compression algorithms to achieve better compression ratios in less time. PGGTC does not specify the compression algorithm and can be used with many. VPC is a value predication-based compression scheme proposed by Burtscher and Jeeradit [9, 25] and has been shown to be very efficient. This approach requires a particular format of the event trace for the predictors so we did not compare our results against theirs or combine PGGTC with VPC.

Various research has proposed to compress the address sequences losslessly [18, 26, 27, 9, 10]. But lossless compression is not practical for larger parallel applications with substantial random access patterns. We show that a performance-wise similar sequence can be generated based on the observed properties of the actual address sequence. This is an extension to the idea of filtering by discarding non-critical addresses to save space [28, 29]. However, filtering techniques discard too much information in the effective address sequence for cache simulations [29]. Toomula [30] proposed using sampling to collect memory skeleton to represent full traces. We also applied sampling techniques [19, 20, 31] in Selective Dumping. However we use sampling technique only for groups for which no static patterns could be detected. In our scheme, the simulator is driven by addresses generated from sampled addresses combined with those sequences which could be represented exactly and compactly.

6 Conclusion

In this paper, we introduced path grammar guided trace compression to efficiently compress control flow traces. Instead of "rediscovering" the patterns one can easily tell from static control flow graph, PGGTC builds the grammar and rules from static analysis of the binary and use these rules to guide online trace compression. PGGTC encodes the paths based on the rules and make them more friendly to generic compression schemes such as gzip. Thus combined with gzip, PGGTC can compress the whole program control flow traces more than 300 times smaller than using gzip alone.

Application	FULL EA trace	Size of SPH	Selective Dumping	Memory Signatures	SyncInfo
BT.A.4	263.5GB	23.9KB	151.8KB	3.6MB	7.6MB
CG.A.4	5.4GB	2.3MB	8.59MB	1.7MB	9.5MB
EP.A.4	7.4GB	9.1MB	25.2KB	25.6KB	469B
FT.A.4	18.7GB	24.4KB	441.5KB	647.5KB	15.5KB
IS.A.4	3.1GB	1.9KB	688.4KB	8.6KB	1.2KB
LU.A.4	158.9GB	17.1KB	105.1KB	3.4MB	5.9KB
MG.A.4	12.6GB	21.8KB	192.9KB	4.7MB	5.9KB
SP.A.4	161.2GB	49.6KB	346.7KB	9.5MB	12.9KB

Table 3. Size of Full trace, Selective Dumped Trace and Memory Signature

Application	Error Using Selective Dumped Traces			Error Using Generated Addresses		
	L1 Error(%)	L2 Error (%)	L3 Error (%)	L1 Error (%)	L2 Error (%)	L3 Error (%)
BT.A.4	1.60	2.15	0.37	2.03	2.00	0.51
CG.A.4	1.34	1.36	0.18	1.50	1.00	0.16
EP.A.4	0.04	0.22	0.16	0.11	0.70	0.16
FT.A.4	2.78	2.64	0.40	2.98	1.65	0.27
IS.A.4	0.50	0.62	0.09	3.15	2.10	0.23
LU.A.4	2.00	1.63	0.18	3.45	2.15	0.54
MG.A.4	2.34	1.75	0.43	3.81	2.95	0.79
SP.A.4	1.87	2.62	0.55	4.25	3.02	0.86

Table 4. Average Absolute Error of Using Approximate Traces

PGGTC is considerable faster than other dictionary based compression algorithms which dynamically create rules based on observed patterns in the trace. Those algorithms have substantial overhead in maintaining dictionaries and looking up rules during compression, and have the danger of ever increasing dictionary as we experienced when using Sequitur to compress EP trace. PGGTC+gzip is on average 40 times faster than widely popular Sequitur algorithm, generating slightly larger trace files. PGGTC can be used with Sequitur also. The combination can generate twice smaller traces than the already highly compressed ones by standalone Sequitur, and can complete the compression 14 times faster.

Effective addresses are larger and more difficult to compress but important for such uses as cache simulation. We proposed Trace Approximation by only keeping compact summaries of random addresses. The summary can either be generated by dumping selectively at given intervals, or by recording the statistical and locality properties of the addresses. Performance similar addresses are generated, not decompressed, from the compact summaries, with the guidance from Structured Path Histories generated from PGGTC. The summaries of memory access takes several order of magnitude less space to store than full effective address traces. In addition, the size of memory access summaries can be estimated and controlled so the user can be sure there is enough disk space for the address traces. Approximated addresses are indeed similar to original traces from the standpoint of cache simulators. By feeding the generated approximated address stream to a cache simulator simulating 25 different cache configurations, we see less than 3% error of absolute percentage difference in cache hit rates acquired by feeding the simulator with complete address traces.

References

[1] K. Skadron, M. Martonosi, D. August, M. Hill, D. Lijia, and V. Pai, “Challenges in computer architecture evaluation,” in *Report on National Science Foundation*, 2003.

[2] J. K. Flanagan, B. E. Nelson, J. K. Archibald, and G. Thompson, “The inaccuracy of trace-driven simulation using incomplete mulitprogramming trace data,” in *MASCOTS*, pp. 37–43, 1996.

[3] X. Gao, M. Laurenzano, B. Simon, and A. Snavely, “Reducing overheads for acquiring dynamic memory traces,” in *IEEE International Symposium on Workload Characterization*, 2005.

[4] X. Gao, B. Simon, and A. Snavely, “ALITER: An asynchronous lightweight instrumentation tool for event recording,” in *Workshop on Binary Instrumentation and Applications (WBIA)*, 2005.

Processor	L1	L2	L3
altix1	16KB, 4way, 256B, LRU	256KB, 8way, 128B, LRU	6MB, 12way, 128B, LRU
Itanium	256KB, 8way, 128B, LRU	3MB, 12way, 128B, LRU	
IT2B	256KB, 8way,128B, LRU	4MB, 16way, 128B, LRU	
O3KA	32KB, 2way, 32B, LRU	8MB, 2way, 128B, LRU	
O3KB,	32KB, 2way, 128B, LRU	8MB, 2way, 128B, LRU	
OPT1	64KB, 2way, 64B, LRU	1MB, 16way, 64B, LRU	
Power3	64KB, 128 way, 128B, random	8MB, 4way, 128B, LRU	
Power4-690b	32KB, 2way, 128B, LRU	720KB, 4way, 128B, LRU	16MB, 8way, 512B, LRU
Power5	32KB, 4way, 128B, LRU	960KB, 10way, 128B, LRU	18MB, 12way, 512B, LRU
Alpha EV67	64KB, 2way, 64B, LRU	8MB, 1way, 64B, LRU	
Xeon5	16KB, 4way, 64B, LRU	512KB, 8way, 64B, LRU	1MB, 8way, 128B, LRU
system 01	32KB, 2way, 128B, LRU	768KB, 4way, 128B, LRU	16MB, 8way, 512B, LRU
system 02	32KB, 2way, 128B, LRU	768KB, 8way, 128B, LRU	16MB, 8way, 512B, LRU
system 03	64KB, 2way, 64B, LRU	1MB, 16way, 64B, LRU	
system 04	256KB, 8way, 128B, random	6MB, 12way, 128, LRU	
system 05	8KB, 4way, 64B, LRU	512KB, 8way, 128B, LRU	
system 06	64KB, 2way, 64B, LRU	8MB, 1way, 64B, LRU	
system 07	128KB, 2way, 64B, PLRU	1MB, 16way, 64B, PLRU	
system 08	128KB, 8way, 64B, LRU	1MB, 8way, 64B, LRU	
system 09	256KB, 8way, 128B, PLRU	12MB, 12way, 128B,NRU	
system 10	64KB, 4way, 128B, PLRU	1920KB, 10way, 128B, PLRU	18MB, 12way, 256, PLRU
system 11	256KB, 8way, 128B, PLUR	6144KB, 6way, 128B, PLRU	
system 12	16KB, 8way, 128B, LRU	2MB, 8way, 64B, LRU	
system 13	16KB, 2way, 32B, LRU	512KB, 4way, 32B, LRU	
system 14	64KB, 4way, 64B, LRU	2MB, 8way, 128B, LRU	

Table 5. Cache Configurations Simulated

[5] D. H. Bailey, E. Barszcz, J. T. Barton, and et. al, "The NAS Parallel Benchmarks," *The International Journal of Supercomputer Applications*, vol. 5, pp. 63–73, Fall 1991.

[6] "http://www.gzip.org/,"

[7] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[8] C. G. Nevill-Manning and I. H. Witten, "Compression and explanation using hierarchical grammars," *The Computer Journal*, vol. 40, no. 2/3, 1997.

[9] M. B. Computer, "VPC3: A fast and effective trace-compression algorithm,"

[10] L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia, "SIGMA: a simulator infrastructure to guide memory analysis," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pp. 1–13, 2002.

[11] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: a transparent dynamic optimization system," *ACM SIGPLAN Notices*, vol. 35, no. 5, pp. 1–12, 2000.

[12] S. Nandy, X. Gao, and J. Ferrante, "TFP: Time-sensitive, flow-specific profiling at runtime," in *LCPC*, 2003.

[13] J. R. Larus, "Whole program paths," in *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pp. 259–269, ACM Press, 1999.

[14] "http://www.zlib.net/,"

[15] A. Srivastava and A. Eustace, "Atom: a system for building customized program analysis tools," in *PLDI '94*, pp. 196–205, ACM Press, 1994.

[16] L. C. Carrington, M. Laurenzano, A. Snavely, R. L. Campbell, and L. P. Davis, "How well can simple metrics represent the performance of hpc applications?," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, p. 48, IEEE Computer Society, 2005.

[17] A. D. Samples, "Mache: no-loss trace compaction," in *Proceedings of the 1989 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp. 89–97, ACM Press, 1989.

[18] E. E. Johnson and J. Ha, "PDATS: Lossless address space compression for reducing file size and access time," in *Proceedings of 1994 IEEE International Phoenix Conference on Computers and Communication*, 1994.

[19] R. E. Kessler, M. D. Hill, and D. A. Wood, "A comparison of trace-sampling techniques for multi-megabyte caches," *IEEE Transactions on Computers*, vol. 43, no. 6, pp. 664–675, 1994.

[20] D. A. Wood, M. D. Hill, and R. E. Kessler, "A model for estimating trace-sample miss ratios," *1991 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems ACM SIGMETRICS Performance Evaluation Review*, vol. 19, no. 1, May 21-24, 1991.

[21] X. Gao and A. Snavely, "Exploiting stability to reduce time-space cost for memory tracing," *Lecture Notes in Computer Science*, vol. 2659, pp. 966–975, 2003.

[22] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, 1998.

[23] Y. Zhang and R. Gupta, "Timestamped whole program path representation and its applications," in *Conference on Programming Language Design and Implementation*, pp. 180–190, 2001.

[24] C. Nevill-Manning and I. Witten, "Identifying hierarchical structure in sequences: A linear-time algorithm," in *Journal of Artificial Intelligence Research*, vol. 7, pp. 67–82, 1997.

[25] M. Burtscher and M. Jeeradit, "Compressing extended program traces using value predictors," in *International Conference on Parallel Architectures and Compilation Techniques*, pp. 159–169, 2003.

[26] Y. Luo and L. K. John, "Locality-based online trace compression," *IEEE Transactions on Computers*, vol. 9, no. 6, pp. 723–731, 2004.

[27] A. Milenkovic and M. Milenkovic, "Exploiting streams in instruction and data address compression," in *IEEEE 6th Annual Workshop on Workload Characterization*, 2003.

[28] A. Agarwal and M. Huffman, "Blocking: exploiting spatial locality for trace compaction," in *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pp. 48–57, 1990.

[29] S. Das and E. Johnson, "Accuracy of filtered traces," in *14th Annual International Conference on Computers and Communications*, 1995.

[30] A. Toomula and J. Subhlok, "Replicating memory behavior for performance prediction," in *Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, pp. 1–8, 2004.

[31] W.-H. Wang and J.-L. Baer, "Efficient trace-driven simulation methods for cache performance analysis," *ACM Trans. Comput. Syst.*, vol. 9, no. 3, pp. 222–241, 1991.