Exploiting Stability to Reduce Time-Space Cost for Memory Tracing

Xiaofeng Gao and Allan Snavely

San Diego Supercomputer Center, University of California, USA
xgao@cs.ucsd.edu, allans@sdsc.edu

Abstract. Memory traces record the addresses touched by a program during its execution, enabling many useful investigations for understanding and predicting program performance. But complete address traces are time-consuming to acquire and too large to practically store except in the case of short-running programs. Also, memory traces have to be re-acquired each time the input data (and thus the dynamic behavior of the program) changes. We observe that individual load and store instructions typically have stable memory access patterns. Changes in dynamic control-flow of programs, rather than variation in memory access patterns of individual instructions, appear to be the primary cause of overall memory behavior varying both during one execution of a program and during re-execution of the same program on different input data. We are leveraging this observation to enable approximate memory traces that are smaller than full traces, faster to acquire via sampling, much faster to re-acquire for new input data, and have a high degree of verisimilitude relative to full traces. This paper presents an update on our progress.

1 Introduction

Research in performance modeling and prediction relies heavily on application traces, especially memory traces. Previous researches have shown that interactions between a program and the memory-hierarchy of the machine on which it executes can largely determine its performance [1,4]. In our own previous work [4], we have shown that summarized memory traces are, to a first approximation, machine-independent; we used memory traces in performance models to predict and explain the performance of scientific applications with different problem sizes using both strong and weak scaling and across several modern High Performance Computing (HPC) platforms. We further used the models to predict the performance of future hardware upgrades and new machines. Thus, given a memory trace it is possible to explain observed cache hit-rates (for example) and to predict cache-hit rates for future machines. It is further possible to guide the tuning of an application, to match that application with machines well suited for its memory demands, and to design future machines towards the needs of the application.

Unfortunately, complete memory traces that are the most perfect representation of a program's memory behavior require Gigabytes of storage. And we found that methods of trading time for space to summarize memory traces (including our own methods for recording predicted cache-hit rates by processing the address stream on-the-

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2659, pp. 966–975, 2003.
© Springer-Verlag Berlin Heidelberg 2003

fly) are not fully satisfactory for three reasons: 1) the time required for summarizing may increase the already severe slowdown for tracing, 2) a summary may reduce the ability to use the same trace to predict performance on a different machine, and 3) this approach generally does not keep continuity of the dynamic execution, so any other analysis, or even the same analysis with minor changes in the parameters requires one to repeat the trace.

Full traces can be compressed rather than summarized to save space. Several researchers [2,8] have successfully used run-of-length, SEQUITUR [10], and other traditional lossless compression techniques such as LZV [16] to reduce trace size. However these compression techniques neither consider nor preserve control flow structure in the trace. Thus the compressed traces do not reflect the structure of the application. It is hard or impossible to get a hint of what will come out from the decompression pipe next, so it is nearly impossible to use techniques such as fast forwarding to reduce analysis cost. Also the compression ratio varies widely depending on the nature of the trace. For a memory trace with mostly random accesses or unpredictable major branches, the compression ratio is quite low and file-size savings minimal [13].

We are developing approximate memory traces that are reasonably small and preserve dynamic execution information. The behaviors of the stable memory instructions are represented by patterns similar to regular expressions. We approximate the behaviors of random instructions with synthetically generated random numbers. Infrequent paths in the control flow may be pruned off to further save space. This approximating method stores smaller traces than methods for memory trace compression when a significant amount of accesses are random. It allows fast-forwarding and preserves a high degree of verisimilitude relative to full traces.

Rubin et.al [8] used the SEQUITUR[10] algorithm to compress memory traces and used the resulting compressed trace to study data layout optimizations. Such lossless compression schemes work well for streams of memory accesses which are regular and have a lot of *exact* repetitions. But there are cases where lossless approaches do not work well. Since many scientific applications touch a lot of memory addresses and contain substantial interspersed randomness, there is not much space saved by using lossless compression techniques on their memory traces. The SIGMA tool [13] from IBM may also adopt similar approaches but with an undisclosed compression algorithm. Judging from their published results, the quality of compression largely depends upon the nature of the application and the input, and the user has little power to control the output size.

In point of fact, trace analysis tools, such as various simulators for the memory sub-system, may not be very sensitive to minor changes in the trace and therefore lossy traces can have satisfactory results. Several researchers have shown that cache simulators can still provide reasonable results when using sampled memory traces as the input [18–20]. However, in these works the sampling rate is a rule-of-thumb. There is no universal rule proposed as to where and how the memory trace should be sampled. Different applications may require quite different sampling rates to remain close to the original trace. Even for the same application, different input may also require quite different sampling rates.

Several other lossy compression schemes have been proposed for particular analysis. Kaplan suggested a lossy reduction scheme for virtual memory simulations [17]. It drops addresses guaranteed to be not visible to virtual memory. This scheme also makes certain assumptions about the hardware. Agarwal and Huffman

[21] suggested a lossy trace compression scheme by exploiting spatial locality in conjunction with temporal locality. All these lossy compression schemes are less than satisfactory when a significant amount of memory accesses are random.

We are looking for a scheme that can find high-level memory access patterns from the trace, yet preserves continuity and enough details for accurate performance prediction. We also want to be able to control the size and accuracy of the trace.

In our previous work, we found it critical to distinguish regular and stable behaviors (constant or clear patterns for memory accesses) from irregular and random ones. Currently, we have found that for the random access areas the trace can be approximated without having noticeably bad effects on subsequent analysis and modeling. Absolute values from the random parts, where generic compression schemes and other lossy schemes fail to have a satisfying compression ratio, turn out to be not too important from a performance standpoint and can be replaced by generated random values. We also observed that several very similar yet unequal sequences in memory traces puzzle generic compression schemes and cause them to fail to have satisfactory compression ratios. But when minor variations in sequences are ignored and lumped together, the compression ratio can be improved by orders of magnitude again without discernable impact on analysis and modeling steps. So when trace size is a concern, similar sequences and random sequences are the best candidates to be approximated. For regular and stable sequences, any generic compression scheme may be used without loss. Based on these observations, we here propose a framework to detect stability in the trace, to classify sequences by similarity and randomness, and to use that information to compress and approximate the trace using various approaches appropriate to each. We show that the tradeoff between the size and the quality of the trace can be controlled by definitions of randomness and similarity.

2 Memory Trace Break-Down

The order of addresses accessed by a program during execution is a function of its dynamic traversal of the control-flow graph *and* the memory access patterns of its individual load and store instructions. As an elucidating example, consider the code fragment in Figure 1.1. We assume the content of array **i** is nearly random. It is difficult for an encoding scheme based on exact pattern detection to summarize the memory access pattern of this fragment. The difficulty arises from two factors. First there is no particular pattern in the effective addresses of array **X**. Although array **i**, **Y** and **Z** are accessed with fixed stride their patterns are defaced in the address stream by **X** due to its random nature. Discernable order in the address stream is further mangled by the branch instruction since the two paths in the loop have each different numbers of memory references. Thus if an encoder simply observes the generated address stream and attempts to detect and encode patterns, it will have difficulty achieving much compression.

However if we focus on individual instructions and study the stride patterns of each instruction the hidden patterns in the stream suddenly becomes clear. Instruction 1, 4, 5 all have fixed stride, while 2,3,6,7 are random. Also, there is a pattern in the order of *instructions* that can be given by the regular expression
(1,4,5,6,7,1,2,3,1,2,3)*. If one random address in the same range is as good (or as bad) as another from a performance standpoint, then reproducing the fixed strides of instructions 1,4,5 and any random values for the addresses touched by 2,3,6,7 along

with the order these instructions are encountered, will serve well enough to represent this fragment's memory behavior.

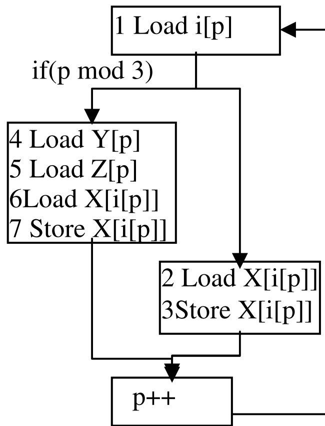

Fig. 1.1. Code Fragment

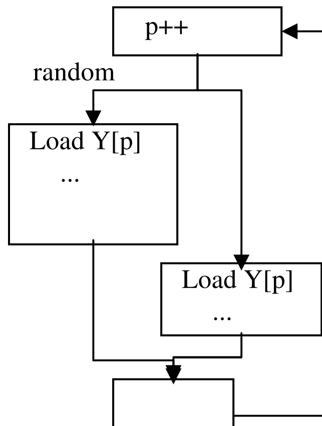

Fig. 1.2. Code Fragment with Randomness

We gather an approximate memory trace by collecting two primary kinds of information. One keeps track of memory instruction ids in the trace and tries to find patterns in order-of-instructions. The other keeps track addresses for individual instructions, and detects stride patterns in these addresses. When no clear pattern is found, these instructions are classified as random—this is a *good* thing from the compression standpoint as we assume their behavior can be usefully mimicked by some random sequence of addresses in the same range. At this time, we should mention that the stride patterns of each instruction typically do not change too much over different input (section 4). It is the order-of-instructions that usually can change a lot depending on input.

When two memory instructions are in the same basic block, their relative order in the trace is fixed: they always appear together with the same instruction distance between them. There is a one-to-one mapping from a sequence of memory instruction in a trace to a path in the control flow graph (we assume for each function call, there is an edge from the calling point to the entry pointer of the callee). So we can use a stream of basic block indices to replace a stream of memory instruction ids. This is particularly important for storing small traces. The number of basic blocks in an application is usually significantly smaller than the number of memory instructions.

3 Detecting Memory Stride Patterns

In our framework, we use different compression approaches for different memory instructions. If an instruction generates effective addresses randomly, we discard the real effective addresses and use some random numbers in the same range to replace them. When an instruction shows clear patterns in the effective addresses, we record

the patterns without loss. One immediate question is how to efficiently detect pattern and randomness of the effective addresses one instruction generates.

Common sense suggests that when a compiler generates a memory instruction, it has a particular functionality; either it accesses a temporary variable, or a data structure in some order. The functionality of the instruction is consistent and stable (which does not mean its access pattern is necessarily regular). This internal functionality determines how the instruction generates the effective addresses, thus the stride between two consecutive addresses for it. For example, the instruction used to incrementally traverse an array often has only two strides: one positive stride equals to the size of the data structure and one negative stride to jump back to the beginning of the array. When the effective addresses generated by one instruction are indeed random, the number of strides must appear large. So the number of strides one instruction generates can be a good approximation of the randomness of the effective addresses of the instruction thus an indication of the nature of that instruction. We set a parameter R to define "randomness". If an instruction is observed to have less than R strides, we classify it as regular and record the patterns without loss. Otherwise it is regarded as random and its effective addresses will be replaced in our traces by a random number generator.

Table 1. Stride pattern categories

	1	2	3	4	5-16	>16
1	1369/ 54.95%	33/ 0.00%	80/ 0.13%	0/ 0.00%	12/ 0.02%	6/ 0.01%
2		82/ 8.18%	30/ 0.06%	9/ 0.00%	2/ 0.10%	4/ 0.00%
3			125/ 3.29%	20/ 0.00%	8/ 0.00%	9/ 0.00%
4				26/ 0.02%	6/ 0.00%	8/ 0.00%
5-16					53/ 7.87%	104/ 0.83%
>16						149/ 24.65%

Because we study stride patterns independent if dynamic control flow, these patterns will be particularly useful if they do not change much over different inputs. Table 1 shows how many instructions change their number-of-strides categories in gzip from SPEC2000 over 5 different reference runs[1].

Entry (i,i) shows the number of instructions that always have i strides for all five runs. Entry (i,j) shows the number of instructions have either i or j strides in the 5 different runs. There are only 20 static instructions that have more than 2 strides, these are counted multiple times in the table. The percentage entries show how much these instructions contribute to the dynamic instruction mix. (The total dynamic instruction count is the *average* of the 5 inputs.)

As can be seen, most of the instructions (static or dynamic) have the same number of strides regardless of input. There are only a few instructions that have different numbers of stride patterns depending on input (entries off the diagonal). In this im-

[1] The inputs of the five reference runs are input.graph, input.source, input.program, input.random, input.log

plementation, we choose R between 4 and 16. The shaded area highlights the maximum number of instructions affected by these choices.

Errors are introduced if we regard an instruction as random when in reality it is not. Figure 1.2 shows an unpredictable branch. On either path, array **Y** is accessed. Because the branch takes the two paths randomly, the stride pattern of the two memory instructions are random in our definition. However, if it is recognized that both load instructions access the same array then it is clear that **Y** is not really accessed randomly. In our implementation two random functions are used to approximate them independently. Although we do not record the absolute values of the random addresses, it will be necessary to study the correlations of the effective addresses generated by the memory instructions in the same loop to reduce these errors. It is also necessary to record statistical properties such as the range one random instruction can access. Using these statistical properties, we can generate more "accurate" random numbers.

4 Application Signatures

An application signature is a compressed dynamic control flow of the application for a given input. It summarizes and approximates the time-varying behaviors of the application on the given input. The pertinent features of signatures include the number of iterations for loops, what paths are taken and how they are taken and how the functions calls behave etc. Application signature provides valuable information about how the program's control flow changes with different inputs. It is can also be used to study correlation between dynamic behaviors and the inputs.

Just as we approximate memory access pattern information, we also approximate dynamic control flows based on stability. We instrument all the basic blocks in the application with DyninstAPI [6]. An online analysis breaks the stream of basic block ids into regions and studies the dynamic behaviors of those regions. For stable regions (with no or very few variations), we keep the exact dynamic behaviors. For unstable regions, we keep only approximated behaviors. The signature of the entire application is composed of a set of signatures of regions and the transition patterns among these signatures. Less significant signatures and similar signatures can be merged to simplify the dynamic control flow, thus reducing the size of the application signatures.

In order to be able to tell the boundaries and study the dynamic behaviors of each individual region, we have an extensive static analysis of the control flow graph before instrumentation. We first find all the loop heads and procedure entry blocks. These basic blocks are called region leaders and are used to mark the boundaries of the regions. The rest of the basic blocks are assigned to the inner-most containing region. Figure 2.2 gives an example of the basic blocks contained in the three regions in Fig. 2.1. The purpose of this assignment is to enable on-line analysis to study the regions' dynamic behaviors independently. For example, the loop headed by block 1 may have very complicated behaviors, but it does not affect the stable behavior of the inner loop headed by 6. Notice if a region is a loop, we do not regard the loop head as part of region. The loop head is included in the outer region as a place-holder to mark some variation of the inner loop happens on that path.

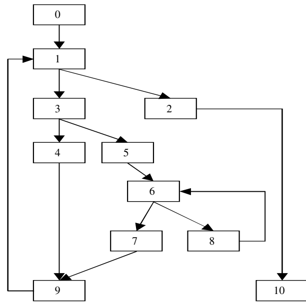

Fig. 2.1. Control Flow with three regions

```
L(0)={0,1,2,10}
L(1)={3,4,5,6,7,9}
L(6)={8}
```

Fig. 2.2. Three regions

A static path in a region is a longest non-repetitive sequence of basic blocks contained in that region. It also must start from the region's leader. For example, the region led by basic block 1 in Fig. 2 has two static paths: 135679 and 1349.

If a static path is set, the number and order of the events that can happen along the path are fixed. The events can be procedure calls, nested loops or memory instructions. This does not mean that the actual events that happen along this path are fixed. A static path just acts as a template and indicates some variations of the events will happen in a deterministic manner along this static path. We use dynamic path to summarize the actual events that happened along this static path. We define a dynamic path as an instance of static path with nesting structures having a signature.

After the static analysis, we instrument all the basic blocks in the application with their region ids. An on-line inspector will then capture a trace of the basic blocks and uses a stack to simulate the entering and exiting of the regions. Upon the completion of each static path, a dynamic path is generated to summarize all the events that happened along the just-passed path. This dynamic path is compared to the previously visited dynamic paths of this static path. If this is a new one, it will be inserted in the dynamic list for later usage. Upon the exit of each region, the inspector generates a signature to summarize the dynamic behaviors of the region in the just-past instance. For example, the summary of a loop structure includes loop iterations, the number of iterations the loop executed, the dynamic paths visited and the transition patterns and distributions among the dynamic paths. The newly generated signature is then compared to previous signatures of this structure and the signature list is updated if there are no same-signatures recorded before. Otherwise the signature index is passed to the outer region to be part of dynamic path information. Finally the application signature is the signature of the outmost region, for example the main() function.

Although it is possible to have the inspector connected to a database and store all the recorded signatures and dynamic paths on disk without any loss, we want to approximate unstable regions using the expected behaviors. In our implementation, each static path has a maximum number of dynamic paths. If there are too many different dynamic paths, the less significant ones will be merged and treated as one dynamic path. Each region also has a maximum number of signatures. Whenever this cap is exceeded, the inspector will choose two similar signatures and merge them. Useful definitions of *similar* are an area of our ongoing investigation. One open investigation is in how to define and compute the similarity distance between two signatures of a structure. In our implementation, we assign different weights to each structure field ad

hoc. For example, the exact transition sequence of dynamic paths within a loop may be less important from a performance standpoint than the distribution of these dynamic paths. So we may call two invocations of the same loop similar if they contain similar distributions of dynamic paths. The distance between two signatures is computed by summing the distances of each field times the specified weights. Another intuition is that it may be useful to only merge the less visited signatures. In future work we plan to describe and explore the utility of various formal definitions of structure similarity.

5 Generating Memory Traces: Analysis of Results

An approximate memory trace is generated by embedding memory stride patterns into the applications signature.

```
(2001,<A4,S4,B0,ST>,(4000:4):1048576),

(1000,(1001,<A5,S4,B0,ST>:65536),

(1003<A4,S4,B0,LD><A7,S4,B0,ST>:1048575),

(1005<A7,S4,B0,LD><A5,RAN,*,LD><A5,IMM0,*,ST>:1048575),

(1007<A5,S4,B8,LD><A5,I-4,*,LD><A5,IMM0,*,ST> :65534), (1009:5)

(1,1000,(1001,<A5,S4,B0,ST>:65536),(1003<A4,S4,B0,LD><A7,S4,B0,ST>:10
48575),(1005 <A7,S4,B0,LD><A5,RAN,*,LD><A5,IMM0,*,ST>:1048575),(1007
<A5,S4,B8,LD><A5,I-4,*,LD><A5,IMM0,*,ST> :65534),(1009:5):10),

7000,10000,11000,6000,3000,
(3001<A5,RAN,*,LD><A5,IMM0,*,ST> <A4,RAN,*,ST><A7,S4,B4,LD>:1048575),

(3008<A4,S4,B8,LD><A4,IMM0,*,LD>:1048573)
```

Fig. 3. Generated full memory trace for IS.W with default input

The simplified approximate memory trace for IS.W is shown in Figure 3. Memory accesses are embedded in the applications signature and demarcated with <>. Each memory access has four fields to specify the data structure to which it belongs, the stride, the base address and the type of the operation. For example <A4,S4,B0,ST> means the memory instruction accesses the array 4 with stride of 4 bytes from the index 0, and it is a store operation. <A5,RAN,*,LD><A5,IMM0,*ST> shows two connected memory instructions. The first one is a random load from Array 5. The second is a store operation, it stores to the exact same address as the previous load. The application signature is demarcated with (). The number after semicolon is the expected iterations of the loop. For conciseness, constant memory accesses, and the jump-back negative strides for loops are removed from this trace.

We used this framework to obtain memory traces of the NPB benchmark suite. Table 3 compares the size of our application signatures and strides pattern information to the size of full memory traces for these benchmarks and gives the error in dynamic instruction count for our approximations. The sizes of *compressed* traces by traditional means would not be much less than full traces for CG and IS because of their

significant amounts of random accesses. Using approximate traces, we can get consistent high "compression" ratios for all these applications.

Table 2. Size comparison and errors

	Estimated memory trace size[2]	Signature Size	Size of the stride patterns	Error in Instruction count	Error in MOPs	Error in FLOPs
IS.S	98MB	5KB	13KB	-0.001%	0.000%	-0.001%
CG.S	898MB	32KB	368KB	0.006%	0.009%	0.006%
FT.S	1718MB	54KB	129KB	0.203%	0.230%	-0.010%
MG.S	125MB	287KB	554KB	2.247%	2.247%	-3.905%
EP.S	1007MB	1KB	5KB	-0.183%	-0.185%	-0.198%
SP.S	3216B	272B	1KB	0.298%	-3.511%	0.000%

Error in total instruction count, as given in Table 2, is only a rough indicator of the verisimilitude of an approximate trace. The real question of interest is how well it mimics an application's true performance behaviors. We used the fast computation methods described in [11] to calculate cache hit rates from the generated IS.W memory trace *without* simulation on three different processors: Power 4, Alpha EV67 and McKinley. For loops with regular stride patterns, such as 1003 and 1007, the calculated cache miss rates and measured ones are indiscernible. For loops having random access patterns, such as 1005 and 3001, the average error in cache hit rates is 2.5%.

6 Conclusion and Future Work

In this paper, we presented a framework to generate approximate memory traces. By dividing the memory trace into stride pattern and application signature, we can differentiate instructions with simple and regular access patterns from complicated or nearly random ones. Different compression or approximation schemes are used for different cases. In this framework, we also proposed methods to make tradeoffs between the trace size and the accuracy of the trace. Initial results have shown that this framework with multiple compression and approximation schemes works fairly well. The memory trace can be approximated and recorded with a small file, yet remain fairly close to the original trace from the perspective of performance estimation. There are still several open questions such as how to compute the distance between two signatures, and the effect of false random instructions on the quality of the overall trace need to be further investigated. The entire scheme needs to be scaled up to an investigation of multiple full scientific applications and we are undertaking that now.
This work was sponsored in part by the Department of Energy Office of Science through SciDAC award "High-End Computer System Performance: Science and Engineering". This work was sponsored in part by a grant from the Department of Defense High Performance Computing Modernization Program (HPCMP) and the National Security Agency.

[2] The full trace size is estimated by the multiplication of the dynamic memory instruction count and the length of the each address.

References

1. Chen Ding and Ken Kennedy "Bandwidth-Based Performance Tuning and Prediction" IASTED, Cambridge, MA November, 1999
2. Richard A. Uhlig, Trevor N. Mudge "Trace-driven Memory Simulation: A Survey" ACM Computing Surveys, V29 No. 2, 1997
3. D.H. Bailey, T. Harris et.al The NAS Parallel Benchmarks 2.0 The International Journal of Supercomputer Applications 1995
4. A. Snavely, N. Wolter, L. Carrington, R. Badia, J. Labarta, A. Purkasthaya, A Framework to Enable Performance Modeling and Prediction Supercomputing 2002
5. ATOM, see http://www.tru64unix.compaq.com/developerstoolkit/#atom
6. B. Buck, J. Hollingsworth An API for Runtime Code Patching The International Journal of High Performance Computing Application, 2000
7. S. P. Reiss, M. Renieris Encoding Program Executions SIGSOFT 2001
8. S. Rubin, R. Bodik, T. Chilimbi An Efficient Profile-Analysis framework for data-layout optimization POPL 2002
9. G. Ammons, J.R. Larus Improving Data-flow Analysis with Path Profiles SIGPLAN conference on programming language design and implementation. 1998
10. C.G. Nevill-Manning, I.H. Witten Compression and explanation using hierarchical grammars. The Computer Journal 1997
11. R.E. Lander, J.D. Fix, and A. LaMarca Cache Performance Analysis of Traversals and Random Accesses SODA 99
12. T. Ball The Concept of Dynamic Analysis ESEC/SIGSOFT FSE 1999
13. Luiz DeRose, K.Ekanadham, Jeffery K.Hollingsworth SIGMA: A Simulator Infrastructure to Guide Memory Analysis, SuperComputing 2002
14. T. Sherwood, E. Perelman, G. Hamerly and B. Calder "Automatically Characterizing Large Scale Program Behaviors" ASPLOS 2002
15. Steve Muchnick, Advanced Compiler Design & Implementation, Morgan Kaufmann, 1997
16. J. Ziv and A. Lempel, “A universal algorithm for sequential data compression” IEEE Transactions on information theory, pp. 337–343, 1977.
17. Scott F. Kaplan, Yannis Smaragdakis, and Paul R. Wilson, "Trace Reduction for Virtual Memory Simulations" SIGMETRICS '99
18. R.E. Kessler, Mark D. Hill, David A. Wood "A Comparison of Trace -Sampling Techniques for Multi-Megabyte Caches " IEEE Transactions on Computers(1994)
19. D.A Wood, M.D. Hill, R.E. Kessier, "A model for Estimating Trace-sampling Miss Ratios" ACM SIGMETRICS Performance Evaluation Review 1991
20. Thomas M. Conte, Mary Ann Hirsch, Wen-Mei W. Hwu "Combining Trace Sampling with Single Pass Methods for Efficient Cache Simulation" IEEE Transaction on Computers 1998
21. Anant Agarwal, Minor Huffman, "Blocking: exploiting spatial locality for trace compaction", Proceedings of the ACM SIGMETRICS 1990