Replicating Memory Behavior for Performance Prediction

Aditya Toomula
PC-Doctor, Inc.
Reno, NV 89502
aditya.toomula@pc-doctor.com

Jaspal Subhlok
University of Houston
Houston, TX 77023
jaspal@uh.edu

ABSTRACT

This paper introduces a method to monitor an application and generate a short synthetic "memory skeleton" program whose memory access pattern is representative of the application. In particular, the application and its memory skeleton should have similar cache behavior on any memory hierarchy architecture. The objective is to quickly estimate the cache performance of an application on any memory architecture by running its memory skeleton. The paper presents and validates a framework for automatic construction of memory skeletons. The approach is based on sampling the address trace of an executing application, summarizing it, and then employing it to generate a synthetic memory skeleton program. The broad goal of this research is construction of "performance skeletons" designed to quickly estimate the performance of a large application in an unpredictable environment. A performance skeleton must also mimic the communication and execution behavior of the application. However, the memory behavior drives the performance of many scientific applications and hence memory skeletons are a critical component of this approach to performance estimation.

1. INTRODUCTION

Prediction of performance of a long running application in a dynamically changing execution environment is a difficult problem. Often the most effective and efficient method of performance estimation is brief monitored execution of code that represents the application. This approach has been explored based on *performance skeletons* [26]. A performance skeleton is a synthetically generated short running program whose execution time always reflects the performance of the application it represents. Hence, simply executing the performance skeleton in a foreign execution environment provides an estimate of application performance in that environment. This skeleton based approach is particularly suitable where availability of resources is unpredictable such as a computation grid with shared nodes and network links. Another compelling scenario is estimation of performance on a simulated future system since full application execution can be prohibitively expensive.

The basic skeleton construction philosophy is that if the performance skeleton executes operations that are representative of application execution, the performance of the skeleton and the application will be similar in any execution environment. Hence, a performance skeleton must capture the execution behavior of the application in terms of computation, communication and memory access patterns, yet execute for a very short time.

Our approach to skeleton construction is to monitor application behavior during execution, summarize it by identifying repeating phases, and then reproduce it as a synthetic skeleton program. Previous work in our group [26] has introduced a framework for automatically creating performance skeletons that capture the communication and computation characteristics. The procedure is outlined in Figure 1. However, memory/cache behavior is the main determinant of performance for many scientific applications. This paper entirely focuses on capturing the memory access behavior. Hence the specific problem addressed in this paper can be stated as follows: *Given an executable application, construct a short running skeleton program whose memory access behavior is representative of the application.* We will refer to such a program as the *memory skeleton* of the application. An application and its memory skeleton should have similar cache performance irrespective of the system memory hierarchy.

The number of memory accesses in a skeleton can only be a small fraction of the full memory trace of an application. In fact, even collecting the entire memory trace of a long running application is not practical because of the overheads. Hence the challenge of building a memory skeleton is generating code that shows the same aggregate temporal and spatial locality as the application, yet is one or more orders of magnitude smaller in terms of execution time and number of memory accesses. Further, the construction of skeleton must be completed in a modest amount of time on typical computer workstations to be practical. The problem is complicated by the fact that the memory behavior of an application can vary a lot between different phases of execution.

The paper is organized as follows. Section 2 discusses memory skeletons and our basic approach in this research. Sec-

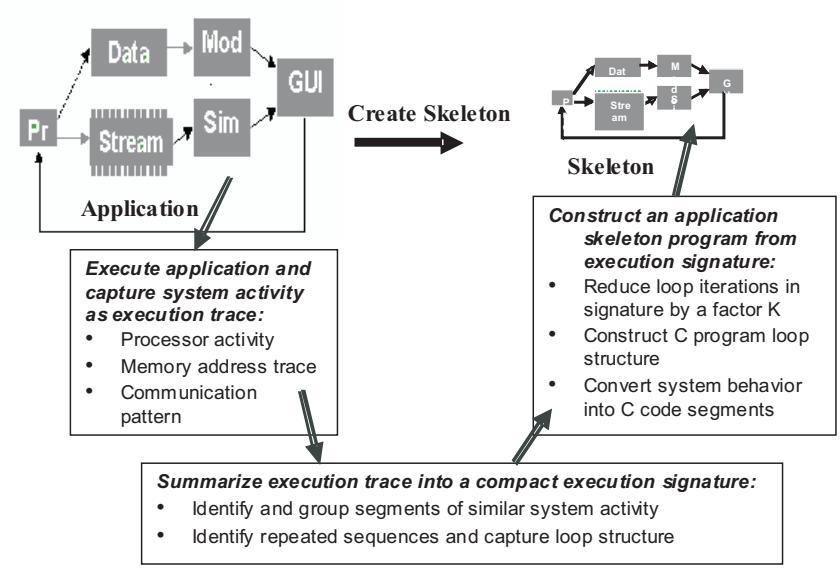

Figure 1: Overview of framework for automatic construction of performance skeletons

tion 3 explains the methodology for constructing and executing memory skeletons. Section 4 presents results that demonstrate the ability of memory skeletons to predict application memory behavior. The remaining sections discuss related work, limitations, ongoing and future work, and conclusions.

2. MEMORY SKELETONS

An application and its memory skeleton should have similar memory reference behavior. A concrete property is that the application and the memory skeleton should have similar cache hit/miss ratio for any cache size. Hence, the skeleton must capture the spatial and temporal data access locality inherent in the application. It is important that a memory skeleton is portable, i.e., the application and the memory skeleton behave similarly across different memory hierarchies and execution platforms.

The starting point for the construction of a memory skeleton is the memory trace generated by the executing application. The approach that we take is to reproduce selected slices of the application memory trace in the generated memory skeleton. However, tracing all memory references requires large amounts of space and time and can be impractical for medium to long running applications. To keep the process manageable, we employ sampling techniques on the memory trace collection itself. The reasoning is that, if samples are taken from a large enough number of points in execution, they provide a statistically sound estimate of the properties of a full execution trace.

Simply reproducing samples of address references across the application does not capture temporal locality, since temporal locality behavior applies only to a consecutive stream of memory references. However, if a sufficiently large block of consecutive references are simulated at every selected program point, temporal locality can be captured. In particular, if the number of consecutive references simulated is an order of magnitude larger than the largest possible cache, the cache behavior is modeled accurately.

The most common performance statistic for memory simulations are the cache miss ratios. The miss ratio is an arithmetic average over time that has been shown to be predictable by statistically sampling the trace [12, 13]. We have taken cache miss ratio as the metric to evaluate how well a memory skeleton represents the application in terms of memory behavior.

Scientific applications often have several diverse phases of execution but show periodicity of behavior at fine granularity within phases. Additional ways to minimize the overhead of skeleton construction are currently being investigated to capitalize on this periodicity. This work is limited to tracking data access patterns (not instructions accesses) which is the main determinant of memory related performance of scientific applications.

3. METHODOLOGY

This section describes the methodology that we have developed to generate the memory skeleton of an application. This methodology is independent of the machine architecture or the way the application is programmed. The three main steps are as follows:

1. Samples of data address trace of the target application are collected as the application is executed on a dedicated machine.

2. Collected traces are compressed into a compact representation.

3. Memory skeleton program is generated from the compact representation of the application trace.

We now discuss each of these steps in more detail.

3.1 Collecting memory address trace

The first step in reproducing memory behavior requires generating memory reference trace of an application. We employ Valgrind Tool [30] to generate a memory address trace. Valgrind is an open source memory debugger for Intel x86 machines running Linux. We have made slight modifications to get the trace of the virtual addresses touched by the application during execution. Valgrind also gives summary cache information. However, this profiling makes programs run 20-100 times slower than normal.

Collecting the whole trace of an application typically involves an unacceptable level of storage space and time overhead. Our trace collection framework based on Valgrind has the ability to capture any number of consecutive memory references, called slices, at any point of execution. It can be periodically switched on and off as desired. The total number of application memory references that are traced determines the overhead of this procedure. The user of this tool can choose to include a small number of large size execution slices or a large number of small size execution slices with similar overhead.

In order to capture the temporal locality behavior exhibited by the application, each trace slice has to be sufficiently long. As a practical rule of thumb, the trace size should be at least one order of magnitude greater than the nominal largest cache size likely to be encountered. "Warming" the cache on present day machines typically takes a few milliseconds or less. If the trace slices are very large, the number of slices is likely to be small for the same amount of overhead, and it is plausible that sample slices don't capture overall execution accurately. The tradeoffs between slice size and number of sample slices will be investigated later in this paper. For the experiments reported in this paper, trace slices collected were uniformly distributed, although other distributions, such as random, can also be employed.

3.2 Trace compaction

The trace collection step above provides a number of slices, each one typically composed of many millions of memory accesses. The size of this trace can be rather large. More important, a crucial engineering problem that would have to be tackled is that of issuing memory accesses at full speed while individual addresses themselves have to be read from a file. To overcome these problems, we "compress" the trace slices using the following two ideas:

1. The exact address in a trace is not critical, i.e., we may generate an address in the skeleton that is a few bytes off of the true address. Addresses that are close to each other are likely to be mapped to the same cache line, and hence variation within that range is not likely to affect memory hierarchy behavior.

2. Slight reordering of the address trace is likely to have a negligible impact on performance. Suppose a reference is made to a given memory location as temporal reference number α in a memory trace. If the same memory access is reference number β when the trace is reproduced, and absolute value of $\alpha - \beta$ is small, in particular if it is negligible as compared to the smallest cache likely to be encountered in a memory hierarchy, then such reordering is not likely to make a significant difference in memory access behavior.

Based on these hypotheses, we proceed with compaction as follows. First, the address space is divided into lines that are the size of a typical cache line. Only the line number to which an address refers is recorded and the exact address is discarded. Next, the memory references (now in the form of line references) in each trace slice are divided into *clusters*, with each cluster consisting of temporally sequential list of line references. The order in which references occur within a cluster is discarded and only the number of references to each line within a cluster is recorded. Note that the original ordering between the clusters is maintained. The *line numbers* and the corresponding access *frequency* is collected and stored in an AVL Tree data structure [28] for each cluster within a trace slice in order. The process is outlined in Figure 2. This compaction reduces the information stored and saves time in generating memory accesses that is discussed later. Reordering within the cluster does alter the aggregate temporal locality but the effect is not significant if the cluster size is small. Our approach to trace compaction is lossy but the results show that this slight temporal reordering still gives accurate predictions.

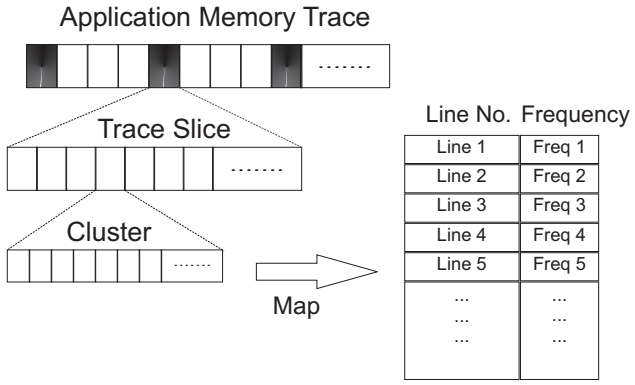

Figure 2: Steps in summarizing memory trace

3.3 Generating the memory skeleton

The goal of this step is to create a C-program that synthetically generates a sequence of memory references recorded in the compacted trace. This seemingly simple step introduces a number of challenges that we discuss here.

- *Minimizing extraneous address references:* An ideal memory skeleton program should generate only the list of memory accesses in the trace. However, any executing program needs memory accesses for its own execution also. If a memory read has to be made to obtain the address of every reference, the number of accesses for bookkeeping will exceed the number of accesses from the trace. This is solved by generating a loop structure for each cluster where reading frequency and line number once leads to a series of actual memory references from the trace without any intervening address reads. The number of extraneous references is not eliminated, but it is reduced to a level that does not impact memory performance significantly.

- *Eliminating cache corruption:* The cache usage of the memory skeleton program should be limited to memory references from the trace that is being simulated. However, when data is read from a file, a memory buffer is created which is mapped to the same cache. This will impact the simulation. To overcome this problem, we use a second machine to read the trace address information from a file and then send it over the network with a sockets interface. The socket buffer is kept very small and does not affect the cache. An alternate approach would be to minimize the buffer size for file reads. However, the approach of using a second machine also nearly eliminates the overhead of file reads as they happen in parallel.

- *Allocating memory* The memory trace is generated as virtual addresses and it is not known a priori what parts of the virtual memory are referenced in the trace. A typical scientific application accesses only a relatively small part of the virtual memory. We have taken a dynamic block allocation approach to address this problem. A substantial size block of memory is allocated when an address reference is made to a location that is not allocated yet. This approach has minimal overhead since memory allocation is done in large size chunks. At the same time the total memory allocated is in the same range as the actual memory that is used by the application since the application itself uses contiguous blocks of memory.

Based on this discussion, the skeleton execution proceeds as follows on a pair of machines, labeled a client and a server. The server machine continuously reads line number-frequency pairs from the compacted trace and sends it to the client. The client machine divides the virtual memory into blocks and maintains a *Sparse Index Table* to see which parts of the virtual memory is allocated. When a new line number-frequency pair is obtained, a higher part of the line number address is used to check if the corresponding memory block is allocated in main memory; if not, then the allocation is made at that point. The lower part of the line number address is used to find the location within the memory block. The resulting location is accessed *frequency* times. This procedure is outlined in Figure 3.

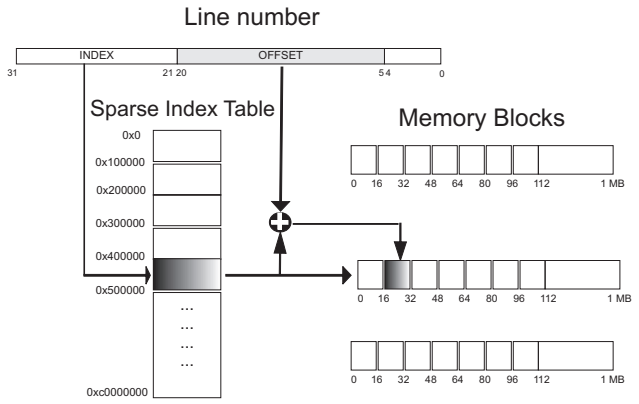

Figure 3: Generation of memory references in the memory skeleton program

4. EXPERIMENTS AND RESULTS

A prototype framework for automatically building memory skeletons has been implemented and skeletons were constructed for several example applications. For validation, cache miss ratios of the constructed skeletons were compared to those of the corresponding applications. The reasoning is that, if the memory accesses in a skeleton are representative of the application, then both should experience similar cache behavior.

4.1 Experimental setup

The construction of memory skeletons and basic validation were done on Intel Xeon 1.7 GHz machines with 256KB 8-way set associative level 2 cache and 64 byte lines. Validation was also done across heterogeneous platforms. The receive buffer size of the client process socket was fixed at 8KB to mitigate its effect on cache behavior.

NAS serial benchmark programs were used for experiments, specifically BT (Block Tridiagonal solver), CG (Conjugate Gradient), IS (Integer Sort), LU (LU solver), MG (Multigrid) and SP (Pentadiagonal solver). These preliminary results are based on Class W NAS benchmarks except for Class A IS benchmark. Memory trace was collected at a fixed sampling ratio of 10% with uniformly distributed trace slices. The size of each trace slice varied across experiments. Trace summarization was done on-the-fly with trace collection to improve efficiency and minimize storage. The size of the *cluster* for compaction was fixed at 256 data references.

4.2 Prediction of cache miss ratios

In the first set of experiments, the cache miss ratio predicted by the skeleton was compared to the actual miss ratio for the application. A minimum trace slice of 10 million memory accesses was used with a minimum of 10 slices across benchmarks. The results are plotted in Figure 4. We note that the skeletons predicted the application miss ratios quite accurately with error within 5%. The lone exception was the IS benchmark which is discussed in more detail in this section.

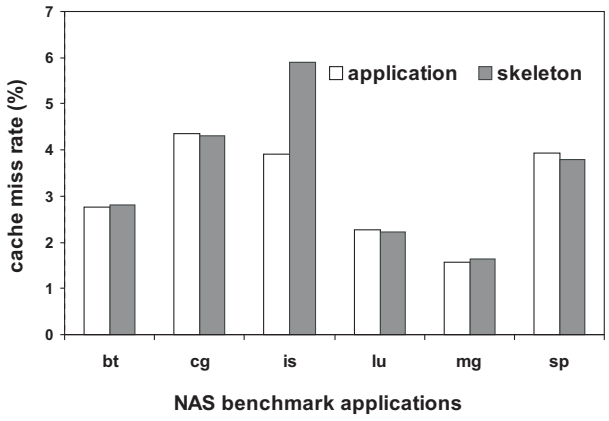

Figure 4: Comparison of data cache miss rate of benchmarks and corresponding memory skeletons

4.2.1 Impact of trace slice selection

We now analyze if the skeleton cache miss ratios depend on which slices happened to be picked during trace collection. For the following experiment, the traces for IS, CG and BT benchmark were divided into 100 uniform slices. Then ten different versions of the memory skeleton were generated for each benchmark - each using 10% of the total trace but a different set of 10 uniformly spaced trace slices. These cases are labeled 1 to 10 in Figure 5 based on the first slice number that was included in the trace.

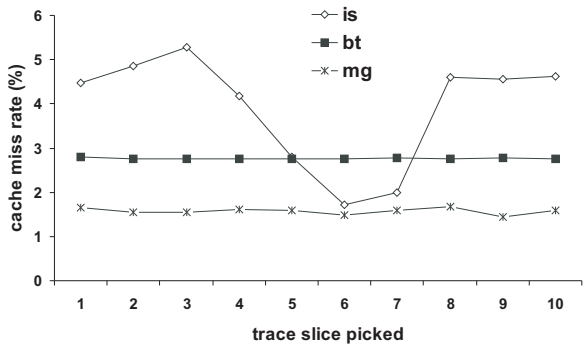

Figure 5: Data cache miss rates for different sets of trace slices in skeletons. Cache miss rates of IS, BT, and MG benchmarks are 3.9%, 2.76% and 1.57% respectively

We observe from Figure 5 that skeletons for CG and BT have similar cache miss rates as the actual benchmarks no matter which slices were chosen. But significantly different cache miss rates were noticed for the IS skeletons. IS (Integer Sort) execution goes through different phases with different memory access behavior unlike BT and CG. In particular, the whole sorting array is accessed at the beginning and at the end of the application execution, thus creating relatively high cache miss rates in those phases. The point is, if all the major execution phases are not proportionately represented in the skeleton of a multi-phase program like IS, then the skeleton can be a poor predictor of the application memory behavior.

One way to make the above scenario less likely is to have a larger number of smaller slices represented in the skeleton. Figure 6 shows the IS skeleton cache miss rates with different numbers of slices and different selected sample slice sets. When the trace was divided into 50 slices, the lowest number used in experiments, there was wide variation in the predicted miss ratio, ranging from nearly 0% to around 6%, depending on which slice set was chosen. As the trace was divided into more slices, the cache behavior showed increasingly consistent results, with most consistent and accurate predictions for the case of 200 slices. Note that in all these cases, the total size of the trace simulated in the skeleton is the same; only the number of slices is varied.

Above discussion indicates that increasing the number of slices increases accuracy. However, the number of slices cannot be increased indefinitely since very small slices do not capture temporal locality accurately. In order to investigate this, we built skeletons for the MG benchmark with different

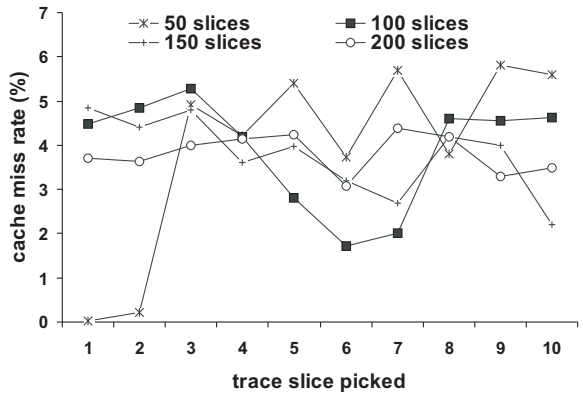

Figure 6: Data cache miss rates of IS skeletons with different sets of slices and for different numbers of slices

sizes of sample slices. The results are presented in Figure 7. We note that the cache miss ratio of the skeleton is within a few percent of that for the corresponding application so long as the slice size is around or above 1 million references. However, if the slices are made smaller, the error increases rapidly.

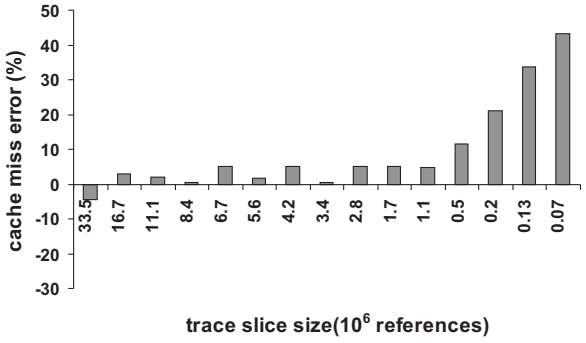

Figure 7: Error in cache miss ratio prediction with memory skeletons for different size trace slices for MG benchmark

4.2.2 Impact of trace sampling ratio

We analyze how the cache miss ratio varies with the ratio of trace samples selected for reproduction in the entire trace. For this experiment, different sample ratios of MG benchmark application trace were collected and memory skeletons were generated for each set of trace sample ratios. The trace slices were uniformly picked throughout the trace.

We observe from Figure 8 that the error in the cache miss ratio prediction of the skeleton increases as the trace sample ratio decreases. However, a higher sample ratio means more of the trace is being simulated implying a higher overhead. A trace sample ratio of around 5% is found to be sufficient to accurately determine the memory behavior of an application. Interestingly, when the trace sample ratio is 100% (i.e., the entire trace is simulated) a small error of 1% is still observed in the cache miss ratio. This is because of the

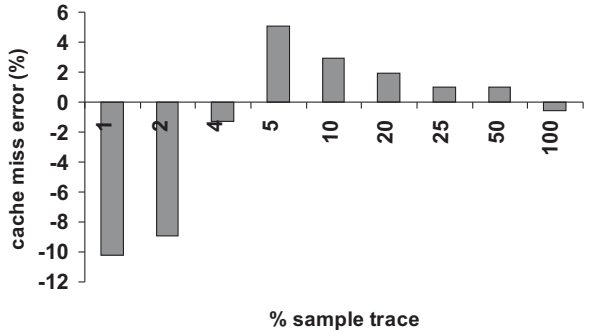

Figure 8: Error in cache miss ratio prediction with memory skeletons for different trace sample ratios for MG benchmark

effect of extraneous address references explained in section 3.3.

4.2.3 *Summary of parameter selection*

A number of parameters have to be selected for building a memory skeleton - the cluster size, the size of trace slices (which then determines the number of trace slices) and the fraction of trace that should be included in the skeleton,. We now discuss what we can conclude about these from our experience. The cluster size should be much smaller than the smallest cache in a hierarchy. We had selected a cluster size of 256 and results confirm that this does not cause significant inaccuracy. The trace slice size should be much larger than the largest cache. Our results show that a slice size above a million is sufficient, above which the benefits of a larger number of slices start outweighing the advantage of larger slices. Finally, not surprisingly, the larger the sampling ratio, higher the accuracy. Generally good accuracy was observed if the sampling ratio was above 5%. We wish to emphasize that these results are for small programs on a typical workstation and more experience is needed for generalizing the recommendations.

4.3 Prediction across hardware platforms

The memory skeletons are designed to predict memory performance of an application across different memory hierarchies and machine architectures. In order to validate portability, we compared the cache miss rate of the same CG skeleton to the CG benchmark on three different machines. First with a 128KB Level-2 4-way set associative cache, second with 256KB, Level-2 8-way set associative cache, and third with a 512KB Level-2 8-way set associative cache. Figure 9 shows the results. The cache miss ratios were predicted fairly accurately with errors within 5% across all machines. Similar behavior was observed for other benchmarks.

5. LIMITATIONS AND EXTENSIONS

This paper is a report on research in progress. The goal is to develop performance skeletons of long running programs that include memory considerations. The specific toolset presented in this paper is able to build memory skeletons for relatively small programs only. The main motive of performance skeletons is to estimate the execution time of the actual application in any execution environment. The present

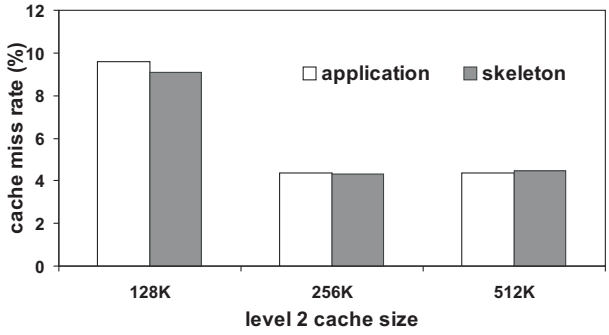

Figure 9: Cache miss comparison of CG benchmark and its skeleton across different memory hierarchies

model of memory skeleton does not predict the application execution time although it can measure cache performance. We list the main limitations of our current system and discuss some of the ongoing and future work toward reaching the goal.

- **Instruction references:** Our present implementation of memory skeletons does not address the instruction references of an application. Memory skeletons only represent the data accesses of the corresponding application.

- **Space overhead:** With the sampling technique employed to collect data addresses, traces are exceeding the capacities of modern storage devices even for traced execution lasting a few minutes. It is possible to use better sampling techniques that explicitly exploit repeating patterns that are observed in the trace to improve prediction accuracy with low space overhead. This is being addressed in ongoing work.

- **Time overhead:** At this stage we are unable to issue the sequences of address references at the full speed of a native machine because of bookkeeping code. One of the reasons is the network latency in transferring the trace summary from server machine to the client machine. We are investigating various steps to reduce the overheads and improve the performance. Note that the above considerations do not affect the cache hit ratios as the bookkeeping code has almost no impact on cache usage.

- **Timing accuracy:** The skeleton execution in the current form switches between execution of bookkeeping code and issue of memory references. To determine the relevant execution time accurately we need a time function which can calculate the clock cycles with nanosecond granularity. The time stamp counter (TSC) can be used to get extremely precise timing information that is theoretically close to nanosecond resolution. However, TSC is available only on Pentium processors . At the time of writing this paper, we are not aware of any portable time function with nanosecond accuracy. *Gettimeofday* function gives time with only microsecond granularity.

- **Integration with communication and CPU events:** Memory behavior modeled in this paper has to be integrated with communication and CPU events to generate complete performance skeletons. This is a future project.

6. RELATED WORK

This research is originally motivated by the problem of estimating performance in dynamically changing grid computing environments as described by Foster et.al. [8]. Condor [14] and LSF [33] address node selection based on CPU and software considerations effectively. Research on node selection based on broader system and network resources has been in the context of getting the best general group of execution nodes [27, 31] or customized procedures to select execution nodes for a particular application or application class [4, 20].

Performance Models: Modeling and analyzing performance of long running applications is an active area of research. Time varying behavior of applications have been studied extensively using different metrics [21]. Phase behavior of parallel applications have been analyzed in [2, 3, 16]. Calder et.al. [22, 23] exploit periodic application behavior to identify portions of the program that are representative of an application for the purpose of micro-architecture simulations. Programs have been evaluated on different metrics and these metrics have been used to design on-line performance prediction models [7]. Reed et.al. [15] use sampling and curve-fitting to model performance with low tracing overheads. Snavely et.al. [24] created application and machine signatures to simulate application behavior across different memory and processor architectures. We have borrowed ideas from these research projects but we employ them to build an executable performance skeleton rather than a performance model [10, 26]. Our earlier work focused on CPU and communication activity. However, performance is often determined by the memory-hierarchy of a machine [6, 25]. This paper is a step toward extending the functionality of skeletons to mimic memory behavior.

Trace Collection and Compaction: Several researchers have successfully compressed memory traces using lossless compression techniques [17, 18, 19, 29, 34]. However, these compression techniques do not preserve the sequence of memory addresses accessed during the execution of an application. Agarwal et.al. [1] introduced a lossy compression scheme by exploiting spatial locality in conjunction with temporal locality. Kaplan suggested a lossy reduction scheme for virtual memory simulations [11]. These lossy compression schemes are less than satisfactory when a significant fraction of memory accesses are random. While trace compression reduces storage costs, it does not reduce simulation time. In contrast, trace reduction seeks to reduce both storage and simulation time by reducing the number of items in the trace. Gao et.al. [9] have presented a framework to speed up the process of collecting memory traces and reduce trace size. We have the taken trace sampling approach, picking multiple samples distributed over the full length of a program's execution. Several research efforts have shown that cache simulators can provide reasonable results when using sampled memory traces as input [5, 12, 32].

7. CONCLUSIONS

This paper has presented a methodology to estimate the cache miss ratio of an application by monitoring, sampling, and reproducing its memory behavior in a synthetic memory skeleton program. We have discussed how the choice of trace slices and trace sampling ratio affect the cache miss ratio prediction accuracy. We have also shown that these skeletons can be successfully deployed to predict memory behavior across different memory architectures. Capturing application memory access patterns in a synthetic program is an important component of a general approach where short execution of a performance skeleton program is used to estimate application performance in a foreign environment such as a dynamic grid or a simulator of a system under design. The paper presents a limited prototype but clearly demonstrates the viability of this approach to building the memory component of performance skeletons.

8. ACKNOWLEDGEMENTS

This research was supported, in part, by the National Science Foundation under award numbers ACI-0234328 and CNS-0410797. Support was also provided by the Department of Energy through Los Alamos National Laboratory (LANL) contract number 03891-99-23 , and by University of Houston's Texas Learning and Computation Center. We wish to thank numerous current and former members of our research group for their contribution to this work. Finally, the paper is much improved as a result of the comments and suggestions made by the anonymous reviewers.

9. REFERENCES

[1] A. Agarwal and M. Huffman. Blocking: Exploiting spatial locality for trace compaction. *Proceedings of the ACM SIGMETRICS conference on Measurement and modeling of computer systems*, 1990.

[2] B. Carlson and T. Wagner. An algorithm for off-line detection of phases in execution profiles. *Computer Performance Evaluation, Modeling Techniques and Tools, 7th International Conference*, pages 253–265, 1994.

[3] B. Carlson, T. Wagner, L. Dowdy, and P. Worley. Speedup properties of phases in the execution profile of distributed parallel programs. *Computer Performance Evaluation, Modeling Techniques and Tools*, pages 83–95, 1992.

[4] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-level middleware for the grid. In *Supercomputing 2000*, pages 75–76, 2000.

[5] T. Conte, M. Hirsch, and W. Hwu. Combining trace sampling with single pass methods for efficient cache simulation. *IEEE Trans. Comput.*, 1998.

[6] C. Ding and K. Kennedy. Bandwidth-based performance tuning and prediction. In *IASTED*, MA, November 1999.

[7] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *International Conference on Parallel*

Architectures and Compilation Techniques (PACT), New Orleans, LA, September 2003.

[8] I. Foster and K. Kesselman. Globus: A metacomputing infrastructure toolkit. *Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[9] X. Gao and A. Snavely. Exploiting stability to reduce time-space cost for memory tracing. *Workshop on Performance Modeling and Analysis - ICCS*, June 2003.

[10] S. Goteti and J. Subhlok. Communication pattern based node selection for shared networks. In *Autonomic Computing Workshop: The Fifth Annual International Workshop on Active Middleware Services (AMS 2003)*, Seattle, WA, June 2003.

[11] S. Kaplan, Y. Smaragdakis, and P. Wilson. Trace reduction for virtual memory simulations. *Proceedings of the ACM SIGMETRICS conference*, 1999.

[12] R. Kessler, M. Hill, and D. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Trans. Comput.*, C-43:664–675, June 94.

[13] S. Laha, J. Patel, and R. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Trans. Comput.*, C-37:1325–1336, Feb 88.

[14] M. Litzkow, M. Livny, and M. Mutka. Condor — A hunter of idle workstations. In *Proceedings of the Eighth Conference on Distributed Computing Systems*, San Jose, California, June 1988.

[15] C. Lu and D. Reed. Compact application signatures for parallel and distributed scientific codes. In *Proceedings of Supercomputing 2002*, Baltimore,MD, Nov 2002.

[16] B. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim, and T. Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE Transactions on Parallel Distributed Systems*, 1(2):206–217, April 1990.

[17] C. Nevill-Manning and I. Witten. Compression and explanation usinh hierarchical grammars. *The Computer Journal*, 1997.

[18] S. Rubin, R. Bodik, and T. Chilimbi. An efficient profile-analysis framework for data-layout optimization. *POPL*, 2002.

[19] A. Samples. Mache: no-loss trace compaction. In *Proceedings of the 1989 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, volume 40, pages 89–97, Oakland, California, United States, May 1989.

[20] G. Shao, F. Berman, and R. Wolski. Master/slave computing on the grid. In *9th Heterogeneous Computing Workshop*, pages 3–16, 2000.

[21] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report 99-630, UCSD-CS, August 1999.

[22] T. Sherwood, E. Perelman, and B. Calder. Basic block-dsitribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2001.

[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, San Jose, CA, October 2002.

[24] A. Snavely, N. Wolter, and L. Carrington. Modeling application performance by convolving machine signatures with application profiles. In *IEEE Workshop on Workload Characterization*, Austin, TX, 2001.

[25] A. Snavely, N. Wolter, L. Carrington, R. Badia, J. Labarta, and A. Purkasthaya. A framework to enable performance modeling and prediction. In *Supercomputing*, 2002.

[26] S. Sodhi and J. Subhlok. Skeleton based performance prediction on shared networks. In *Proceedings of the 4th IEEE Symposium on Cluster Computing and the Grid (CCGrid'04) Workshop on Grids and Advanced Networks (GAN'04).*, Chicago, Illinois, April 2004.

[27] J. Subhlok, P. Lieu, and B. Lowekamp. Automatic node selection for high performance applications on networks. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 163–172, Atlanta, GA, May 1999.

[28] A. Toomula. Construction of memory skeletons for performance prediction. Master's thesis, University of Houston, Houston, TX, December 2004.

[29] R. Uhlig and T. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29(2), 1997.

[30] Valgrind. http://valgrind.kde.org.

[31] J. Weismann. Metascheduling: A scheduling model for metacomputing systems. In *Seventh IEEE Symposium on High-Performance Distributed Computing*, Chicago, IL, July 1998.

[32] D. Wood, M. Hill, and R. Kessler. A model for estimating trace-sampling miss ratios. *ACM SIGMETRICS Performance Evaluation Review*, 1991.

[33] S. Zhou. LSF: load sharing in large-scale heterogeneous distributed systems. In *Proceedings of the Workshop on Cluster Computing*, Orlando, FL, April 1992.

[34] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transaction on information theory*, pages 337–343, 1977.