EMPS: An Environment for Memory Performance Studies

Jeffrey K. Hollingsworth
University of Maryland
hollings@cs.umd.edu

Allan Snavely
San Diego Super Computing Center
allans@sdsc.edu

Simone Sbaraglia, K Ekanadham
IBM T.J. Watson Research Center
{sbaragli, eknath }@us.ibm.com

Abstract

This paper describes an overview of Environment for Memory Performance Studies (EMPS). EMPS is a framework to allow different data gathering and simulation tools to be composed together to predict the performance of parallel programs on a variety of current and future High End Computing (HEC) systems. The framework seeks to combine the automated nature of direct execution simulation with the predictive capabilities of performance modeling.

1. Introduction

The first-order goal of High End Computing (HEC) systems is to provide enhanced time-to-solution for important scientific applications. By leveraging Moore's Law for the regular increase in processor speeds, recent evolution has resulted in systems comprised of large numbers of processors together capable of delivering peak floating-point rates in the 100s of Teraflops range. At the same time, technological limitations and economic factors have caused the latency between processors and memories, as counted in processor cycles, to steadily increase. In an attempt to ameliorate these increasing latencies, memory hierarchies are getting deeper—more levels of cache. In a parallel trend, increased density of memory technology, as well as more sophisticated busses and interconnects are being leveraged in memory hierarchies that are becoming wider—memory subsystems are growing both in size and in the number of processors that share them. As the complexity of the data storage and data movement facilities of machines increases there is a growing tension between writing applications that have good communication performance versus writing applications that have good memory system performance.

Tuning the application for good performance in one of these dimensions can be challenging, while finding the performance sweet spot between the two can be daunting. Likewise, it is not clear that architectural choices, for example whether to spend more money and effort to make the interconnect faster versus making the memory subsystem bigger or faster, can be made today with confidence since the application performance impacts these choices. In light of the difficulty and importance of understanding such tradeoffs, this project is focused on fundamental research, as well as innovative development, in techniques for improving understanding of the performance implications of deep and wide memory hierarchies. We are developing an integrated framework to permit performance modeling of tradeoffs such as the above.

1.1 Importance of performance modeling

The performance of an application is a function of (at least) the implementation, the target architecture, the compiler, the runtime system, the O/S, and the effect of contention for shared resources. A performance model is a calculable expression explaining and quantifying why an application performs as it does on a machine.

A useful performance model is parameterized to allow for "what-if" investigations. As an example of a "what-if" investigation, if a memory performance model has architecture parameters one can then use it to predict how cache hit-rates would change due to larger or more associative cache. On the other hand, if a memory performance model has application parameters, one can use it to predict how cache hit-rates would change if a different cache-blocking factor were used in the application. Performance models have historically been used to improve architecture design, inform procurement, and guide application tuning.

Unfortunately, the process of producing performance models has historically been very laborious and requires large amounts of time and expertise. These challenges have limited the number of HEC applications that have been thoroughly performance modeled. Users of unstudied applications and workloads (the majority), have been somewhat disenfranchised with respect to their ability to influence future architectures, to identify the best machine for their application, and to carry out guided tuning. In recent times it has been observed, due to the difficulty of developing performance models for new applications, as well as the increasing complexity of new systems, that computers have become better at predicting and explaining natural phenomena (such as the weather) than at predicting and explaining the performance of themselves or other computers!

2. Background

We have been working to make performance modeling automated or at least semi-automated. Tools for automated performance model extraction can relieve model developers of much of the time-consuming work, and even some of the expertise needed to produce pa-

rameterized performance models of applications. The ideal tool set would allow a user to follow a recipe by taking some measurements of an application, supplying some information about a target machine (real or hypothetical) and then conducting a rapid simulation to reveal factors affecting performance. Such a systematic approach enables wider ranging studies of larger applications and more workloads with a resulting increase in understanding of the computational demands of applications. We have been developing techniques to efficiently store detailed memory traces, tools for extracting application signatures that are machine-independent summaries of the computational demands of applications, and convolution methods that are time-tractable computations mapping the one to other in order to yield performance predictions [2]. Along the way, substantial evidence has been gathered showing that, due to trends cited above, the interaction of an application with a machine's memory subsystem is often one of the dominant factors explaining its performance. Unfortunately, for reasons described next, this is also often one of the hardest and most time-consuming portions of a model to construct, even when using an automated or semi-automated process. Therefore there is a critical need for research and development in this area.

3. EMPS

We are developing Environment for Memory Performance Studies (EMPS). It contains components implementing different functionality, which can interact to assist the task of memory performance analysis. EMPS is an extensible environment, one can conceive that each functionality may have many different implementations, which is indicated by showing multiple instances for each box in Figure 1. In the subsequent sections we describe how the various components interact. In the future, the investigators and other researchers and vendors should be able to come up with new designs of boxes with the same functionalities or new boxes with other functionalities that may be added to the environment. The environment presents many choices for designers of applications and architectures, to experiment with various techniques and analyze the resulting performance. A designer can choose a path through selected boxes from Figure 1 and drive them to examine the results at the end. Thus, EMPS provides a highly flexible and extensible environment for researchers and vendors to provide different tools and users to experiment with different combinations of tools suitable for their pursuit.

In order for the EMPS environment to be flexible and extensible, the interfaces (or the API) between the

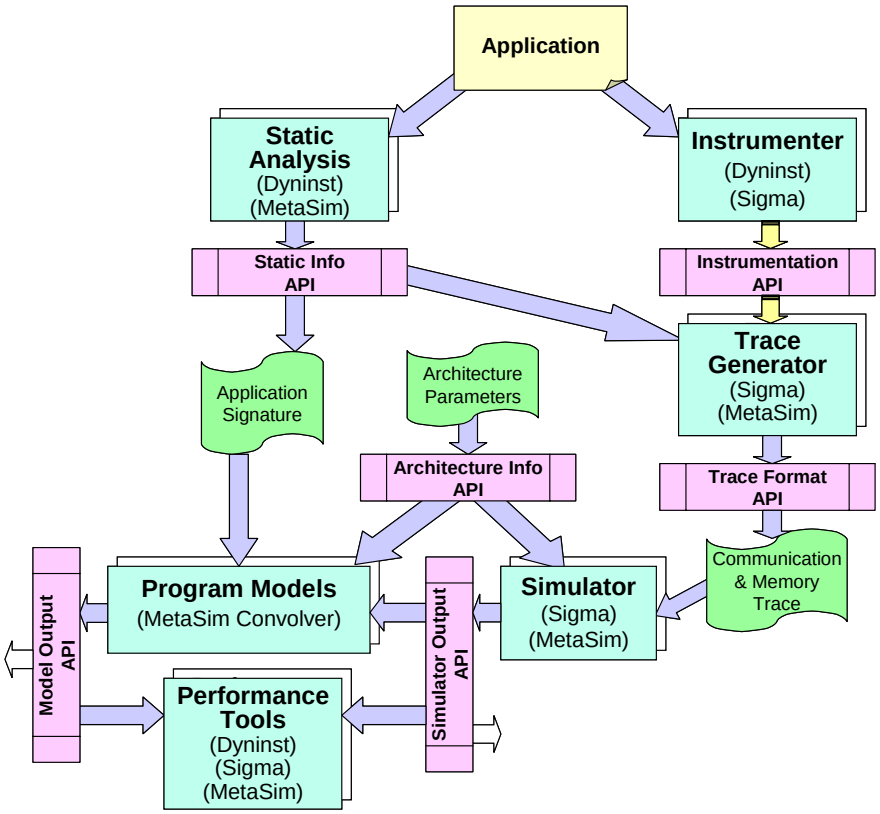

Figure 1: Instantiation of EMPS with current implementations.

components must be well-defined and adhered to, so that coupling of different components happens in a smooth manner. Towards this goal, we are working to define the APIs shown in Figure 1. Vendors should be able to supply their implementations without revealing proprietary details. Users must be able to couple the components without need to understand the internal details of the components. We next describe some of the existing techniques that are being adapted to create EMPS.

3.1 Dyninst

To address practical difficulties for gathering application traces, Dyninst API [5], a portable interface for acquiring performance data, has been developed. Dyninst allows for dynamic instrumentation, de-instrumentation, re-instrumentation of a running program to obtain performance counter information, and additional information including memory addresses generated. Dyninst permits starting a long-running HEC application normally, letting it run for a time, and then attaching instrumentation instructions to its program memory (the program now slows down due to instrumentation overhead) to observe and *sample* its performance behavior, then detach. When instrumentation is disabled, the program returns to normal behavior and speed. This process can be repeated at regular or dynamically determined intervals.

We plan to develop methods leveraging this technology to gain sufficient coverage of an application, and sample enough information about factors affecting its performance, to obtain usefully accurate application signatures enabling accurate performance models without slowing down the application too much. Section 3.4 provides some preliminary results on this effort. Also, there is an opportunity to ameliorate the Heisenberg effect by both perturbing the program less than would be the case by full tracing, and by being able to iteratively compare performance between instrumentation events to that expected from data gathered during instrumentation events, thus to infer the perturbation due to instrumentation.

In [34] we showed early evidence that this approach could be useful for applications performance modeling. We also uncovered many important open research questions with respect to the perturbation due to instrumentation, the effects and accuracy of sampling, and the tradeoff between sampling interval and resulting signature fidelity.

3.2 Sigma

We have also been working to develop Sigma [11], a software infrastructure to guide memory analysis. Sigma was developed at IBM Research to meet internal and customer demands for tools to assist in reasoning about the performance implications of ever more complex memory subsystems. The main goals of Sigma are: 1) provide feedback to programmers to identify bottlenecks, problems, and inefficiencies in a program due to mapping of data structures into the memory hierarchy, 2) provide suggestions for performance improvements, 3) predict performance on current and hypothetical systems, and 4) provide feedback on design of new architectures. As opposed to traditional control-centric performance tools, Sigma has also a data-centric framework, providing an environment to help users to identify data structures and code segments that are causing poor program performance, without requiring re-execution of the application under analysis.

The Sigma environment consists of a pre-execution tool that locates and instruments all instructions that refer to memory locations, a runtime data collection framework that performs a highly efficient lossless compression of the stream of memory addresses generated by the binary instrumentation, and a family of simulation and analysis tools that process the compressed memory reference trace to provide programmers with tuning information. The simulation and analysis tools include a TLB simulator, a data cache simulator, a data prefetcher simulator, and a query mechanism that allows users to obtain performance metrics and memory usage statistics.

This comprehensive simulation environment has special emphasis on techniques for modeling memory hierarchies and is accessible and customizable via APIs. In order to simulate advanced features of emerging shared-memory architectures, We are enhancing Sigma's capabilities to model event-ordering, false-sharing and cache-to-cache transfers. Also, we are working to use Dyninst based trace sampling to improve the time required to obtain a memory trace of large HEC applications.

3.3 MetaSim

In order to be able to automatically extract models of an application's memory behavior and to map that to an arbitrary memory hierarchy, MetaSim Tracer and MetaSim Convolver [33] have been developed. The MetaSim tracer is a tool built on top of Dyninst API. It extracts performance information including memory access patterns from each basic block of an instrumented program during a tracing run and saves a (small) file summarizing that information. The MetaSim Convolver is a tool that takes as input a MetaSim trace file and combines it with a machine profile that includes information about a target machine's memory hierarchy (gathered, for machines that exist with the PMaC MAPS benchmark [33], or generated synthetically or by simulation for machines on the drawing board). By combining these two inputs the convolver generates a perform-

ance prediction. It thus functions as a statistical simulator, combining statistics from applications and machines; it is much faster than cycle-accurate simulators; but it is still potentially quite accurate. In [7 , 33, 34] we showed substantial evidence that this approach can usefully predict the performance of a range of applications across dimensions of input, processor count, and target machines including for strong and weak scaling. We also uncovered many important research questions with respect to the machine independence of memory access patterns, tradeoffs between signature compactness and fidelity, performance similarity of signatures (do signatures that look almost the same perform almost the same?), as well as needed technological advances needed to allow modeling of innovative memory technologies such as PIM (Processor in Memory) that are beyond just incremental changes to today's memory hierarchies.

3.4 Trace Sampling

Another technique we have been investigating is the idea of trace sampling to reduce the volume of memory traces being generated. For many scientific applications it is important to understand the memory system in great detail, yet an analysis for a relatively few timesteps might be sufficient to understand the overall application's performance. In [7] we showed early evidence that the performance characteristics of several scientific applications could be determined by 10% or even 1% sampling of all dynamic memory references. We have been working to develop several techniques that will allow traces to be gathered for representative subintervals of a program's execution.

To allow trace sampling, we will use the Dyninst system to enable and disable trace collection. Using Dyninst it is possible to temporarily remove instrumentation from a program. We plan to eventually make this technique more flexible by adding extensions to Dyninst to allow disabling instrumentation on a per-function basis (currently instrumentation can only be enabled or disabled globally). To provide some indication of the data that was missed by disabling trace sampling, we will leave a small amount of instrumentation enabled to record such things as the number of timesteps that were omitted from the trace.

Using Dyninst provides the mechanism for controlling trace generation. However, a critical question is what policy will be used to decide when to turn trace sampling on and off. Since timesteps are a natural unit for work for many scientific programs, we use them as an initial granularity of trace sampling. Although its possible to identify timestep boundaries by looking at the program alone, we currently have the application programmer indicate either the name of a function that is invoked for each timestep, or alternatively a file and line number where the timestep starts. With this information, we can insert instrumentation at that point in the program to record the timestep boundaries.

4. Multi-processor Memory Hierarchies

One of the most difficult problems in developing new parallel applications, particularly ones that use irregular meshes or adaptive mesh refinement (AMR), is to define an efficient domain decomposition;; the developer has to map the logical structure into the parallel system topology. The parallelization has to deal with the determination of sub-domains and their mapping to the processors. Currently, the main techniques for domain decomposition are based on distributing these subdomains such that the computation load is uniform and the communication requirements during the parallel computation are reduced. However, a distribution that generates low communication requirements will not necessarily have an efficient utilization of the memory subsystem. Hence, it is important for programmers to model and understand the dynamic relationship between data access patterns and communication that will occur during the execution of a program, but there are no mechanisms that allow one to evaluate how different data partitioning will behave with regards to communication and utilization of the memory subsystem.

The goal of this research thrust is to enable "what if" investigations by varying parameters of machines and/or applications. A particular focus is to be able to model how demands of the interconnect, and demands of the memory subsystem, vary as the data decomposition and problem size changes. This will enable tradeoff analysis of, for example, using more processors to move each processor's working set up in the memory hierarchy, or fewer processors to reduce communications overhead and will allow users to find the sweet spot for this tradeoff. It will also enable architectural design guided by demands of applications by, for example, indicating whether communications technology or memory subsystem technology is the bottleneck for a set of modeled codes.

We are developing techniques for including multiprocessor effects in machine profiles. Today systems are built with single chips that contain multiple processors and multiple levels of caches around them. Since communication delays play an increasingly dominant role in the design, multiple paths are employed to transfer information between the caches and processors. This leads to a new set of problems in the study and analysis of memory hierarchies. Some of the specific issues we will address: *Data placement, Event-Ordering, and False-Sharing*.

4.1 Data Placement

We will carry out research to develop performance measurement, analysis, and modeling infrastructure for the understanding of the interaction between communication intensity and pressure on the memory subsystem, such that one could evaluate the performance of a program for different domain decompositions. Our simulators will be extended to support parallel architectures, (both distributed memory and shared memory systems), and their most common program paradigms (MPI and OpenMP, respectively). In addition, we will capture communication related data, which will require research to extend the trace compression mechanisms, as well as the trace representation. By capturing data references associated with parallel execution, the simulator can predict the behavior of the program when elements are redistributed. Hence one will be able to compare and contrast the performance for different distributions, and highlight how new access behavior is influenced by redistribution.

4.2 Event-Ordering

Most of the prior analysis of caches [12] has focused on uni-processor systems. While there are some multi-processor performance studies [37], modeling caches for them will necessarily involve modeling timing interactions between concurrent executions. For instance, when two processors share a cache, a cache model must be driven by a trace of spliced accesses made by the two processors, where the splicing is determined by the speeds of the two processors. Hence generating individual traces of memory references made by each processor is insufficient for this purpose and capturing the timing interaction between the processors is much harder as it might require detailed cycle-by-cycle simulations. For practical performance studies, one needs to come up with some simple approximation strategies for studying this problem.

We will instrument parallel programs so that traces are collected for each stream of instructions that can be executed potentially concurrently with others. In addition we will capture the synchronization events executed by these streams that determine their ordering at these points. Using this information, one can model various scenarios for splicing the streams.

4.3 False-Sharing

When two independent streams manipulate data that happens to be mapped into a single cache line, conventional cache protocols can cause a ping-pong effect, known as false-sharing. When a processor executing one stream updates its part of the cache line, the line is evicted from the cache used by a second processor using the other part of the cache line and the scenario is reversed when the second processor updates the line. For instance Berger et al [3] report this to be a serious problem and proposes clever dynamic memory allocation schemes that alleviate this problem. For statically allocated data structures, one may resort to proper padding to avoid placement of unrelated items that get mapped to the same cache line. In order to identify such hot spots, one needs to have mechanisms that detect false sharing patterns.

The basic instrumentation will capture concurrent streams and their corresponding memory traces, our simulations can then highlight potential false-sharing instances and relate them to source data structures so that application designers can be alerted. By providing a means to predict the cache miss behavior under selected padding, one can also verify the fixes for the problem. Our infrastructure will also permit performance predictions under altered memory allocators. That is, we will predict the cache miss behavior when a different memory segment is allocated for a dynamically allocated data structure. Thus, it will be possible to identify when different schemes would alleviate the false-sharing problem.

5. Summarizing performance behaviors

We desire compact representations of the resource demands of applications to specify exactly what an application will ask a given machine (existing or hypothetical) to do on its behalf. We call such representations *application signatures.* Note that the source code of an application could be considered a high-level description, or application signature, of its resource demands. However, depending on the language it may not be very compact (Matlab is compact, Fortran is not compact) while determining the resources demands of the application from the source code may not be very easy; in fact this could require actually compiling the code and running it (or simulating it if the machine does not exist). Hence we need cheaper, faster, more flexible ways to obtain representations suitable for performance modeling investigations. A minimal goal is to fold the results of several compilation, execution, performance data analysis cycles into a signature so these step do not have to be repeated each time a new performance question is asked.

A dynamic instruction trace, including a record of each memory address accessed, such as can be acquired with our Dyninst based tools Sigma and MetaSim could also be considered an (very detailed) application signature. But it is not compact (address traces alone can run to Gigabytes for even short running applications) and it is not machine independent. We described some approaches to reducing the size of raw traces in the previous section.

The basic approach to acquiring application signatures is to conduct series of experiments where pro-

grams are traced with the techniques from the previous section, and the traces are further analyzed by pattern detection. Recurring sequences of messages and loads/stores are represented by patterns. Important sequences of patterns are detected and expressed as signatures. Infrequent paths through the program are ignored. Patterns that map to insignificant performance contributions are dropped. As a very simple example, the performance behavior of CG, the Conjugate Gradient benchmark from the NAS Parallel Benchmarks, with source code of more than 1000 lines, can be represented *from a performance standpoint* by one random memory access pattern. This is because 99% of execution is spent in the following loop:

```
do k=rowstr(j),rowstr(j+1)-1
        sum = sum + a(k)*p(colidx(k))
enddo
```

This loop has two floating-point operations, two stride-1 memory access patterns, and one random memory access pattern (the indirect index of p). On almost all of today's deep memory hierarchy machines the performance cost of the random memory access pattern dominates the other patterns and the floating-point work. Practically, to predict the performance of CG on a machine all that required is the size of the problem (which part of the memory hierarchy it fits in) and the rate at which the machine can do random loads from that level of the memory. Thus a random memory access pattern succinctly represents the most important demand that CG puts on any machine.

We are working to extract just the important performance features of codes, and to make these as succinct and machine independent as possible to enable efficient modeling. This approach will deal with the complexities that arise with full applications that spend a significant amount of time in more than one loop or function. For example, in many applications, one pattern does not dominate a given loop and patterns must be combined and weighted. Also, simple addition might not be the right combining operator for patterns because the machine can overlap different types of work (say memory accesses and communications). Also, our framework needs to consider the impact of different compiler or different flags that result in better code (so trace results are not machine independent).

6. Related Work

Execution-driven simulators [16, 36] were first developed for MIPS processors. Subsequently, Torrellas *et.al*., [35] extended MINT to collect timing characteristics of Intel processors, which was ported to PowerPC processors (Augment6k) by Giampapa [13]. All these simulators were concerned with timing characteristics of program segments. They trapped memory access instructions and transferred control to a backend that can simulate desired memory architecture. Non-memory access instructions were run on the native processors and efficient techniques of estimating their timing were incorporated.

There have been some tools that access hardware performance counters. For Intel platforms, Vtune[17] is available. PAPI [4]provides a multi-platform interface to access hardware counters. However, these approaches only provide counters of data or sampling among code regions. In contrast, our work provides detailed information about individual memory references, **and** the actual memory addresses being accessed.

Other systems have taken advantage of the flexibility provided by the hardware to add instrumentation of datacentric caches. ATUM [1] uses the ability to change the microcode in some processors to collect memory reference information. The FlashPoint [22] system used the fact that the Stanford FLASH multiprocessor [19] implements its coherence protocols in software, allowing instrumentation to be added at this level. Buck and Hollingsworth [6] proposed using interrupt on overflow to sample the addresses of data cache misses, but this approach does not provide the level of detail provided by EMPS. Also, our work is designed to allow "what-if" questions that can not be answered by using direct measurement alone.

Mtool [14] provides information about the amount of performance lost due to the memory hierarchy, but only relates this information back to program source lines, not to data structures. A system with more similarity to the techniques in this proposal is MemSpy [21], which provides data-oriented information as well as code-oriented. StormWatch [8] is another system that allows a user to study memory system interaction. It is used for visualizing memory system protocols under Tempest [28], a library that provides software shared memory and message passing. However, the goal of StormWatch is to study how to adapt a memory system protocol to suit the application, rather than how to change the application to match the memory system. Because of this, the information provided is also different. This information includes what protocol events are taking place, what code is causing them, and how they are related.

The SimPoint project [27] provides tools for picking simulation points so that the performance of a full program can be extrapolated from some sample-simulated intervals. Haskins and Skadron [15] developed fast-forwarding and check pointing to enable incremental simulation of full applications over time. Lafage and Seznec [20] give methods for automatically finding where to simulate for reasonably approximate program verisimilitude. Statistical sampling approaches

are explored by Conte et.al [9]. Statistical simulation whereby program traces are generated to represent the behavior of whole programs via much compressed synthetic representation is reported in Nussbaum and Smith [26]. No one of these systems by themselves provide the full capabilities of EMPS.

In pioneering modeling work Saavedra [29-31] proposed to model applications as a collection of independent Abstract Fortran Machine tasks. These simple models worked well on the simpler processors and shallower memory-hierarchies of the mid 90's.

For parallel system predictions Mendes [24, 25] has proposed to record the explicit communications among nodes and to build a directed graph based on the trace. Then sub-graph isomorphism is used to study trace stability and to transform the trace for different machine specifications. Simon [32] proposed to use the Concurrent Task Graph to model applications.

Crovella and LeBlanc [10] proposed complete, orthogonal and meaningful methods to classify all the possible overheads in parallel computation environments and to predict the algorithm performance based on overhead analysis. Xu, Zhang and Sun [38] proposed a semi-empirical multiprocessor performance prediction scheme. Their general ideas for performance prediction from partial measurements of orthogonal properties have influenced our work.

The performance-modeling group at LANL has developed highly successful models for predicting application and machine performance [18, 23]. However, their techniques require extensive paper and pencil analysis to derive models.

7. Conclusions

In this paper we have described our goals and early work on a system call EMPS that provides a plug and play simulation environment for studying memory hierarchies on High End Computing systems.

Acknowledgements

This work was supported in part by NSF awards CNS-0406336 and CNS-0406312.

References

1. A. Agrawal, R. L. Sites, and M. Horowitz, "ATUM: A New Technique for Capturing Address Traces Using Microcode," *Proceedings of the 13th Annual International Symposium on Computer Architecture*. June 1986, pp. 119 -127.
2. D. Bailey, "Performance Evaluation Research Center,".http://perc.nersc.gov/.
3. E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A Scalable Memory Allocator for Multithreaded Applications," *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*. November 2000, pp. 117-128.
4. S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, "A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters," *Proc. SC'2000*. November 2000, Dallas, TX.
5. B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching," *Journal of Supercomputing Applications*, **14**(4), 2000, pp. 317-329.
6. B. Buck and J. K. Hollingsworth, "Using Hardware Performance Counters to Isolate Memory Bottelenecks," *SC'2000*. Nov. 2000, Dallas, TX, IEEE.
7. L. Carrington, A. Snavely, X. Gao, and N. Wolter, "A Performance Prediction Framework for Scientific Applications," *ICCS Workshop on Performance Modeling and Analysis (PMA03)*. June 2003, Melbourne.
8. T. M. Chilimbi, T. Ball, S. G. Eick, and J. R. Larus, "StormWatch: A Tool for Visualizing Memory System Protocols," *Proceedings of Supercomputing '95*. December 1995, San Diego, CA.
9. T. M. Conte, M.A.Hirsch, and K.N.Menezes, "Reducing state loss for effective trace sampling of superscalar processors," *1996 International Conference on Computer Design (ICCD)*. October 1996.
10. M. E. Crovella and T. J. LeBlanc, "Parallel Performance Prediction Using Lost Cycles," *Proceedings of Supercomputing '94*. Nov. 14-18, 1994, Washington, DC, IEEE Press, pp. 600-609.
11. L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia, "SIGMA: A Simulator to Guide Memory Analysis," *SC'2002*. Nov. 2002, Baltimore.
12. J. Edler and M. Hill, "Dinero-IV Trace-driven Uniprocessor Cache Simulator,".http://www.cs.wisc.edu/~markhill/DineroIV.
13. M. Giampapa, *Augmint6k: The Augmint multiprocessor simulation toolkit for IBM PowerPC architecture*, IBM Internal Report.
14. A. J. Goldberg and J. L. Hennessy, "Performance Debugging Shared Memory Multiprocessor Programs with MTOOL," *Supercomputing'91*. Nov. 18-22, 1991, Albuquerque, NM, pp. 481-490.
15. J. Haskins and K.Skadron, "Minimal subset evaluation: Rapid warm-up for simulated hardware state," *International Conference on Computer Design*. September 2001.
16. S. Herrod, *Tango lite: A multiprocessor simulation environment"*, Stanford University.
17. Intel Corporation, "VTune Performance Analyzer,".http: //developer.intel.com/software/products/vtune/index.htm.
18. D. J. Kerbyson, A. Hoisie, and H. J. Wasserman, "A Comparison Between the Earth Simulator and AlphaServer Systems using Predictive Application Performance Models," *in Proc. Int. Parallel and Distributed Processing Symposium (IPDPS)*. April 2003, Nice.
19. J. Kuskin, *et al.*, "The Stanford FLASH Multiprocessor," *21st International Symposium on Computer Architecture*. April 1994, Chicago, IL, pp. 302-313.
20. T. Lafage and A.Seznec, *Choosing representative slices of program execution for microarchitecture simulations: a preliminary application to the data stream*, in *Workload Characterization of Emerging Applications*. 2000, Kluwer Academic Publishers.

21. M. Martonosi, A. Gupta, and T. Anderson, "MemSpy: Analyzing Memory System Bottlenecks in Programs," *1992 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. June 1-5, 1992, Newport, Rhode Island, pp. 1-12.
22. M. Martonosi, D. Ofelt, and M. Heinrich, "Integrating Performance Monitoring and Communication in Parallel Computers," *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. May 1996, Philadelphia, PA.
23. M. Mathis, D. J. Kerbyson, and A. Hoisie, "A Performance Model of non-Deterministic Particle Transport on Large-Scale Systems," *in Proc. of Int. Conf. on Computational Science (ICCS)*. June 2003, Melbourne, Australia, Springer Verlag, vol.part 3, Vol. 2659, pp. 905-915.
24. C. L. Mendes and D. A. Reed, "Integrated Compilation and Scalability Analysis for Parallel Systems," *IEEE PACT*. 1998.
25. C. L. Mendes and D. A. Reed, "Performance Stability and Prediction," *IEEE / USP International Workshop on High Performance Computing*. 1994.
26. S. Nussbaum and J. E.Smith, "Modeling superscalar processors via statistical simulation," *International Conference on Parallel Architectures and Compilation Techniques*. September 2001.
27. E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder, "Using SimPoint for accurate and efficient simulation," *ACM SIGMETRICS Performance Evaluation Review*, **31**(1), 2003.
28. S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Typhoon and Tempest: User-Level Shared Memory," *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*. April 1994.
29. R. H. Saavedra and A. J. Smith, "Analysis of Benchmark Characteristics and Benchmark Performance Prediction," *ACM Transactions on Computer Systems*, **14**(4), 1996, pp. 344-384.
30. R. H. Saavedra and A. J. Smith, "Measuring Cache and TLB Performance and Their Effect on Benchmark Run Times," *IEEE Transactions on Computers*, **44**(10), 1995, pp. 1223-1235.
31. R. H. Saavedra and A. J. Smith, "Performance Characterization of Optimizing compilers," *IEEE Transactions on Software Engineering*, **21**(7), 1995, pp. 615-628.
32. J. Simon and J.-M. Wierun, "Accurate Performance Prediction for Massively Parallel Systems and its Applications," *Euro-Par*. 1996, vol.Vol II, pp. pp675-688.
33. A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha, "A Framework for Application Performance Modeling and Prediction," *SC2002*. November 2002, Baltimore.
34. A. Snavely, X. Gao, C. Lee, N. Wolter, J. Labarta, J. Gimenez, and P. Jones, "Performance Modeling of HPC Applications," *ParCo*. October 2003, Dresden.
35. J. Torrellas, M. Lam, and J. Hennessy, "False Sharing and Spatial Locality in Multiprocessor Caches," *In IEEE Trans. on Computers*, 1994, pp. 651-663.
36. J. Veenstra and R. Fowler, "MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors"." *Proceedings of the Second International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*. January--February 1994, Durham, NC, pp. 201-207.
37. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. 1995, pp. 24-37.
38. Z. Xu, X. Zhang, and L. S. Semi, "Empirical Multiprocessor Performance Predictions," *Journal of Parallel and Distributed Computing*, **39**, 1996, pp. 14-28.