

Effectiveness of Simple Memory Models for Performance Prediction

Irina Chihaiia Thomas Gross
Departement Informatik
ETH Zürich
CH 8092 Zürich, Switzerland

Abstract

Many situations call for an estimation of the execution time of applications, e.g., during design or evaluation of computer systems. In this paper we focus on large applications where the execution times heavily depend on the performance of the memory system. Since such applications are computationally expensive, direct simulation is not an option and an analytical model is called for.

This paper addresses this problem by developing and evaluating two simple analytical models. These models focus on an application's interaction with the memory system. Applications are characterized by their memory access types. A regular application has continuous and strided memory accesses. An irregular application has three memory access types: continuous accesses, accesses within the same L1/L2 cache line, and random accesses.

The analytical models are combined with results from micro-benchmarking or with appropriate performance estimates of memory accesses to predict application performance, either on real or future machines. We apply these models to executions of CHARMM (Chemistry at HARvard Molecular Mechanics) - a scientific application written in FORTRAN, SMV (Symbolic Model Verifier) - coded in C, and NS2 (Network Simulator) - coded in C++. For all three applications, the approaches described here produce results with 5% accuracy on average (compared to the effective run-time measured on a real SPARC system).

1. Introduction

Research and engineering of computer systems often requires to estimate the execution time of programs. For instance, research into paging techniques for large applications [1, 2] or into compressed-memory [3, 4] requires the investigation of applications that have large data spaces and long execution times. However, a detailed simulator that models all aspects of modern processors requires sev-

eral hours to simulate a few seconds of real execution time with reasonable accuracy [5]. Straightforward extensions of existing simulators incur a high computational costs and may not even be practical if the data space is large. As researchers use larger and larger applications (and move away from application kernels) in their simulations, the situation will get worse, despite improvements in the cycle time of the platform that is used for simulations.

An attractive approach is analytical modeling to predict an application's performance on a (real or hypothetical) system. Although analytical models can provide results in short time, most researchers have had limited success in validating their results on real machines. The difficulty of modeling current systems comes from the fact that modern microprocessors use a number of techniques to overlap cache misses with computation and subsequent memory operations. Some of these techniques may have a negligible impact on the performance of some applications, but they may be important for other applications. This poses a significant challenge in developing an accurate performance model with a small set of input parameters that are easy to measure or estimate. As we enlarge the class of applications whose performance we want to predict, the model becomes more and more complex. Validation with a real system, however, is important to increase the confidence that estimations for hypothetical systems will be meaningful.

1.1. Contributions

This paper focuses on performance prediction of applications whose performance depends primarily on the performance of the memory system. We classify applications by their memory access patterns, and we focus on their non-continuous accesses. An application is called *regular* if more than 80% of its non-continuous accesses are strided. We call an application *irregular* if more than 80% of its non-continuous accesses are random. Any application that does not fall into these categories is called mixed application.

We describe two models to estimate the execution time of these two application classes: the MSP-RA (Memory System Performance for Regular Applications) model predicts the run time of regular applications, which are dominated

This research was supported, in part, by a gift from the Microprocessor Research Lab (MRL) of Intel Corp. and by the NCCR "Mobile Information and Communication Systems", a research program of the Swiss National Science Foundation

by continuous and strided accesses. For irregular applications, memory performance is given by the performance of continuous accesses, accesses within the same L1/L2 cache line, and random accesses. These parameters are combined in the MSP-IA (Memory System Performance for Irregular Applications) model that predicts the run time of irregular applications.

The estimation of a program's execution time is based on its number of memory accesses at *each* level of the memory system and on the time it takes to execute these accesses. We measure application and machine performance separately and then combine the two measurements in a simple linear model. We gather a program's memory accesses using an instruction-level trace generator. A full system simulator could provide more accurate/detailed information. However, the huge slowdown of this method and the fact that we study complete, real, complex programs, make this method unpractical. We use either micro-benchmarking to measure the memory hierarchy performance or obtain those data from the system designer (that may very well use a more detailed simulator to obtain a characterization of the memory system). The benchmark used captures a wide range of processor optimizations, such as whether a copy transfer is done in parallel or in sequence. Both read and write times are considered separately. We have validated the usefulness of these models on an UltraSPARC-II system. The results show that these models capture an application's characteristics and effectively map them into a machine characterization.

1.2. Background

We use the term *target* to refer to the system that is to be simulated. At the least we want to estimate the execution time of an application on this target system. The *platform* is the system that hosts our simulator, e.g., the system that executes the application and produces inputs for a simulator. We group related research into three categories: (1) approaches to model the execution of the simulated program, (2) approaches to characterize the target machine, and (3) approaches to characterize the simulation on the platform (i.e., how is the platform coupled to the simulator). These categories are described in more detail in the following subsections.

1.2.1. Performance Prediction

Researchers have proposed a large number of performance prediction techniques, ranging from pure mathematical models to full system simulators. As our goal is to predict performance for large applications, we review only the prediction techniques that use data from executions of real benchmarks.

According to [6, 7, 8], the execution time of a program on a specific machine can be computed by summing up the timings for elementary instructions scaled by their frequency

during a specific execution. The main drawback of this approach is the high amount of measurements it requires. For example, Saavedra and Smith [8] consider 109 abstract FORTRAN operations and measure their execution time on each machine. The technique has been applied only to unoptimized FORTRAN codes; our method works for C/C++ and FORTRAN and handles optimized as well as unoptimized codes.

The profile-based evaluation technique used by Ofelt and Hennessy [9] computes the execution time of a program by summing up the execution times of the basic blocks (and paths) on a given machine times their frequency during a specific run. This technique requires detailed object code analysis and/or compiler assistance to identify the basic blocks of a program. Our work focuses on performance prediction for (large) complex applications and, for simplicity, we avoid detailed program analysis.

Hennessy and Patterson [10] state that a good measure of the memory hierarchy performance (although still an indirect measure of performance) is the average time to access memory. The Average Access Time (AAT) is defined in terms of time to hit in the cache/memory, miss penalty and miss rate for reads and writes. However, as the processors get more and more sophisticated, characterizing memory system with a single parameter is insufficient and another measure is called for.

Ailamaki et al. [11] compute the execution time of a DB query by summing up the computation time, memory stall time, branch misprediction overhead, and resource-related stalls. By considering the type of memory accesses, we reduce the number of events to be measured/simulated. However, our method requires additional support to determine the access type information.

Most of the current performance models include the program's miss penalty. The number of cache misses can be either measured [7] or estimated using compile-time prediction [12] or a mathematical framework [13].

Recently, researchers have focused on analytic evaluation of shared-memory systems with ILP processors [14, 15]. The proposed models use complex formulas with a large number of parameters (10-20 parameters). We focus on the application and develop a much simpler model for two specific types of programs: regular and irregular codes.

1.2.2. Machine Characterization

Micro-benchmarking is a popular technique used as a basis for performance prediction. Initial work in this area focused on sequential programs and did not consider the memory hierarchy [8]. Since then, the benchmarking technique has been extended to measure performance of parallel machines [16].

Measuring the memory performance of modern systems is difficult, given that not all misses incur the same timing penalties, and misses interact with one another. Hris-

tea et al. [17] classify misses into: (1) in-isolation misses, (2) back-to-back misses, and (3) pipelined misses. In-isolation and back-to-back misses represent the best and worst-case performance of the memory systems, and they dictate the performance of irregular applications. For regular codes, when pipelined transfers are possible, bandwidth is more important than latency [17, 18, 19, 20]. Hristea et al. describe three benchmarks to measure the performance of the three miss categories (for reads only). Similar micro-benchmarks are McVoy’s *lmbench* [20] that measures pipelined and back-to-back memory latency and McCalpin’s STREAM [21] that gathers the memory pipeline performance.

Stricker and Gross [22] describe *memperf*, a bandwidth-oriented characterization of a memory system that pays attention to memory access patterns. Memory performance is measured in access bandwidth for different strides and different working sets. The stride parameter (with values between 1 and 192) shows how well cache and external stream logic help with read ahead. The working set parameter captures the effect of cache hits through reuse of recently accessed data. The write tests capture the performance of well pipelined writes through a write-back queue.

In this paper, we use *memperf* to measure the memory performance [23]. However, the models described here are not tied to this specific benchmark. Other benchmark suites such as Hristea et al. [17] and *lmbench* [20] can also be used to gather the performance numbers. When discussing the target machine, we report how these benchmarks match.

1.2.3. Application Characterization

Conte and Gimarc [24] classify the run-time data collection methods as either hardware-assisted or software-only schemes. The hardware schemes involve the use of hardware devices that are added to a system solely for the purpose of data collection. The most frequently used hardware method is a hardware-based counter that can keep track of events such as cache misses. The software schemes use the existing hardware to gather the desired information. The software schemes are classified into two approaches: those which instrument the application code and those which simulate, emulate, or translate it.

Uhlig and Mudge [5] classify the trace collection methods according to the level of system abstraction where they are extracted. They define eight levels of system abstraction: the circuit, microcode, emulation, loader, linker, assembler, compiler and operating system level.

We characterize an application by counting the number of its memory accesses and their type (size) at each memory level. The access type is extracted from the program address trace. To gather the trace, we use Sun’s *shade* instruction-level trace generator. To process the address trace, we use the *cachesim5* simulator that is distributed with the *shade* tool [25].

2. Two Simple Models for Execution Prediction

Initial work on performance prediction characterizes memory performance with a single parameter, the access latency. For instance, the AAT model [10], briefly discussed in Section 1.2.1, predicts a program’s performance by multiplying the number of its cache hits and misses with the hit/miss times. However, for many applications, the compilers are able to use pipelining and arrange data accesses in an optimal manner. Hence, for these applications, many memory accesses have an access time lower than the average time. On the other hand, the applications whose memory accesses can not be overlapped, access memory with a latency higher than the average value.

In this paper, we restrict the set of programs we model (to regular and irregular applications) and we characterize memory performance with more than one parameter. We now describe the models in more detail; after presenting the key parameters for a real target machine, we then compare the estimates produced by these models with the estimates of the AAT model.

The execution time of a program is the sum of the times the program spends at each memory level. In other words,

$$T = \sum_{i=1}^n T_{total_i} \quad (1)$$

where T_{total_i} is the total time the program spends at the level i of the memory system.

The time spent at a memory level is a linear combination of the number of times each access type appears at that level multiplied by the time it takes to execute that memory access. Clearly,

$$T_{total_i} = \sum_{j=1}^s N_{ij} \times T_{ij} \quad (2)$$

where i is the memory level we consider, N_{ij} is the number of times the access type j appears at the level i , and T_{ij} is the time it takes to execute a memory access of type j at the memory level i .

Putting it together, the execution time of a program is

$$T = \sum_{i=1}^n T_{total_i} = \sum_{i=1}^n \sum_{j=1}^s N_{ij} \times T_{ij} \quad (3)$$

where i iterates over the memory levels (e.g., L1, L2, DRAM) and j over the access types. Both read and write times are considered separately.

The MSP-RA model uses the Memory System Performance for Regular Applications to predict the performance of a regular application. An application is called regular if 80%+ of its non-continuous accesses are strided. (The rest of 20%- can be indexed array accesses.) The performance of regular codes is given by the performance of continuous and strided accesses. (Regular accesses are also called affine array accesses [26].) As regular codes have two kinds of accesses ($s = 2$ in Eq.(3)), the run-time of a regular application is

$$T = \sum_{i=1}^n T_{total_i} = \sum_{i=1}^n \sum_{j=1}^2 N_{ij} \times T_{ij} \quad (4)$$

where N_{i1} and N_{i2} are the number of continuous and strided accesses at the level i of the memory system. T_{i1} and T_{i2} are the time to execute a continuous and strided access at same memory level i .

The MSP-IA model uses the Memory System Performance for Irregular Applications to predict the run time of an irregular application. For this applications, memory performance is given by the performance of continuous accesses, accesses within the same L1/L2 cache line, and random accesses. (Random accesses are also known as pointer-chasing accesses [26].) As irregular codes have three kinds of accesses ($s = 3$ in Eq.(3)), the run-time of an irregular application is

$$T = \sum_{i=1}^n T_{total_i} = \sum_{i=1}^n \sum_{j=1}^3 N_{ij} \times T_{ij} \quad (5)$$

where N_{i1} , N_{i2} and N_{i3} are the number of continuous accesses, accesses within the same L1/L2 cache line and random accesses at the level i of the memory system. T_{i1} , T_{i2} and T_{i3} are the time to execute a continuous access, an access within the same cache line and a random access at the same memory level i .

3. Target System

To allow a meaningful evaluation of the models presented here, we use as a target system a Sun Blade 100 workstation since a variety of good tools are available in source form. (Availability of the sources allowed us to obtain additional parameters of memory accesses, like the strides.) This system has a 16 KB L1 data cache (32 byte blocks, direct mapped), 256 KB L2 cache (64 byte blocks, 4-way set associative), 256 MB DRAM, and is based on an UltraSPARC-II processor that operates at 500MHz.

The memory performance is measured in access bandwidth¹ for different strides and working sets. The performance of continuous accesses (pipelined bandwidth) is the memory throughput when *memperf* performs continuous loads or stores (with stride 1). The performance of strided accesses is the average over the memory throughputs when data is accessed with a small stride (with values between 2 and 7). (Small strides capture the performance of overlapped transfers.) The performance of accesses within

¹Since *memperf* reads doubles (8 bytes), we use the following formulas to express the transfer performance in cycles/sec or seconds: $Perf_{Cycles} = \frac{8bytes \times clockfreq}{bandwidth}$ or $Perf_{Seconds} = \frac{8bytes}{bandwidth}$. For example, on a Sun Blade 100 (500 MHz), the measured L1 cache pipelined bandwidth (continuous reads) is about 2648 MB/s. Hence, on this machine a sequential 8-byte load from the L1 cache needs about 1.5 clock cycles

or 3ns: $\frac{8bytes \times (500 \times 10^6 \frac{cycles}{s})}{2648 \times \frac{10^6 bytes}{s}} = 1.5cycles$ or $\frac{8bytes}{2648 \times \frac{10^6 bytes}{10^9 ns}} =$

$\frac{8 \times 10^3 ns}{2648} = 3ns$

the same L1/L2 cache line is the average over the memory throughputs when data is accessed with stride sizes that fit into L1/L2 cache. (The prefetch effect of gathering a whole line increases the hit rate of accesses within the same cache line.) The average over the memory throughputs when data is accessed with a large stride (ranging from 12 to 192) gives the performance of random accesses. (Large strides defeat the aggressive overlap of cache misses supported by many microprocessors [27].) McCalpin's STREAM also gathers the memory pipelined bandwidth. McVoy's *lmbench* measures the pipelined bandwidth and random read latency using linked-list pointer chasing.

Table 1 lists the measured performance of the Sun Blade 100 for different types of memory accesses at different levels of the memory hierarchy. For this system, the working sets with the characteristics performance of the L1 cache, L2 cache, and DRAM are 16KB, 256KB, and 8MB.

3.1. Data Accuracy

We use McCalpin's STREAM benchmark to partially validate the performance data measured with *memperf*. The copy performance of the Sun Blade 100 measured by STREAM is 83 MB/s. The load copy mode of the *memperf* benchmark provides the same information as the STREAM copy operation. The measured performance of a continuous load followed by a continuous store when *memperf* exceeds the working set of the L1/L2 caches on a Sun Blade 100 is 83 MB/s, which agrees with the number reported by the STREAM benchmark.

4. Sample Applications

4.1. SMV (Symbolic Model Verifier)

SMV is a model checking tool based on Binary Decision Diagrams (BDDs) used for formal verification of finite state systems. SMV is a complex C program that has high CPU and memory requirements. Because it uses dynamically allocated data structures (trees) SMV has an irregular access pattern. We use Yang's SMV implementation [28] since it shows superior performance over other implementations [29]. We select 8 inputs, 6 of them are models used in benchmark studies [29] and the other 2 inputs model the FireWire system [30]. Table 2 presents the memory and run time characteristics of the selected SMV models.

4.2. CHARMM

CHARMM [31] is a macromolecular simulator that has been optimized to fully benefit from the L1/L2 cache performance. Written in FORTRAN, CHARMM accesses its statically allocated data (arrays) in a regular fashion (strided and indexed array accesses). (Indexed array accesses are irregular accesses but they are not sequential as the pointer-chasing accesses. [26]) We select 8 different input files

L1 cache		L2 cache		DRAM		Access Type
Read	Write	Read	Write	Read	Write	
memory system performance for regular applications						
1.51	2.73	5.24	3.58	19.14	32.89	continuous
1.55	2.79	4.92	2.75	57.86	123.44	strided
memory system performance for irregular applications						
1.51	2.73	5.24	3.58	19.14	32.89	continuous
1.54	2.76	6.08	4.16	37.01	73.96	same L1 cache line
2.89	4.18	16.31	27.23	101.05	197.48	random

Table 1. Sun Blade 100: read and write performance [cycles] for different types of memory accesses.

SMV model	Run time [s]	Memory[MB]
semaphore	0.32	43.55
counter	0.30	43.62
pci3p	1.01	44.12
dme1	1.87	47.77
abp8	16.45	79.87
node-3-3-2	24.29	160.34
idle	35.55	159.96
node-3-3-3	71.87	174.93

Table 2. SMV models: memory and run time characteristics on Sun Blade 100.

that are distributed with the CHARMM package. Table 3 presents the memory and run time characteristics of the selected CHARMM inputs.

CHARMM input	Run time [s]	Memory[MB]
gener	1.01	58.22
nbond	1.15	58.22
im	1.32	58.22
brb	1.42	58.23
dyn1	1.83	58.27
ener	2.04	58.24
imh2o	2.20	58.25
djs	6.11	58.25

Table 3. CHARMM inputs: memory and run time characteristics on Sun Blade 100.

4.3. NS2 (Network Simulator)

NS2 is a discrete event simulator targeted at networking research [32]. Written in C++, NS2 accesses the memory in an irregular fashion. We select 4 different inputs (network protocol models) that are distributed with the NS2 package. Table 4 presents the memory and run-time characteristics of the selected NS2 inputs.

4.4. Data Accuracy

An application is characterized by the number of its memory accesses and their type (size) at each memory level. To gather the memory access type, we extended the

NS2 test	Run time [s]	Memory[MB]
diffusion	7.12	14.82
shadowing	69.49	15.13
aodv	102.71	16.68
tdma	126.64	90.40

Table 4. NS2 inputs: memory and run time characteristics on Sun Blade 100.

cachesim5 cache simulator distributed with the *shade* tool. We use the performance counters to partially validate the measured data. Although the hardware counters can gather the number of read/write cache hits/misses, they can not provide the access type information. The events we measure are the number of reads and writes from/to the L1 and L2 caches. Validating the *shade* tool against the performance counters shows an error range of $\pm 10\%$ for the data gathered by *shade*.

5. Experimental Results

5.1. The MSP-RA Prediction Model

The performance of a regular application is computed by the MSP-RA model, given by Eq.(4), which on the Sun Blade 100 workstation becomes

$$T = \sum_{j=1}^2 (N_{L_1j} \times T_{L_1j} + N_{L_2j} \times T_{L_2j} + N_{Mj} \times T_{Mj}) \quad (6)$$

where $j = 1$ denotes continuous accesses and $j = 2$ strided accesses.

Figure 1 shows that the MSP-RA model succeeds to capture CHARMM's characteristics and effectively maps them to the machine characterization (with an error range of $\pm 10\%$).

For the same CHARMM inputs, the AAT model overestimates the real execution time, as shown in Figure 2. Hence, for this application, AAT is a too conservative characterization of the memory system. CHARMM's access pattern consists of array references. The processor overlaps many of these accesses, which shortens the application's run time.

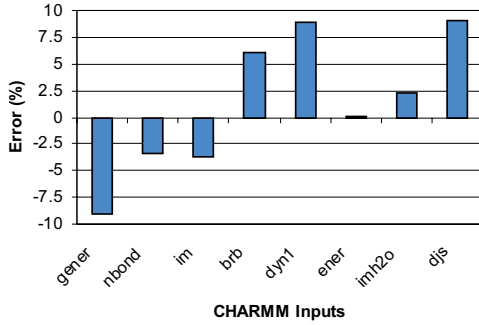


Figure 1. MSP-RA prediction error.

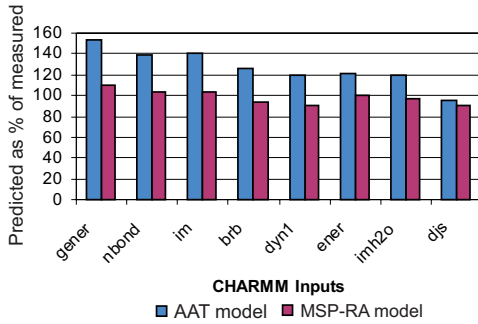


Figure 2. AAT and MSP-RA estimates.

5.2. The MSP-IA Prediction Model

The performance of an irregular application is computed by the MSP-IA model, given by Eq.(5), which on the Sun Blade 100 workstation becomes

$$T = \sum_{j=1}^3 (N_{L1j} \times T_{L1j} + N_{L2j} \times T_{L2j} + N_{Mj} \times T_{Mj}) \quad (7)$$

where $j = 1$ denotes continuous accesses, $j = 2$ accesses within the same L1/L2 cache line, and $j = 3$ random accesses.

Figure 3 and 5 show that the MSP-IA model accurately predicts the performance of the selected SMV models and NS2 inputs (with an error range of $\pm 10\%$).

For the same SMV models and NS2 inputs, the AAT model underestimates the real execution time, as shown in

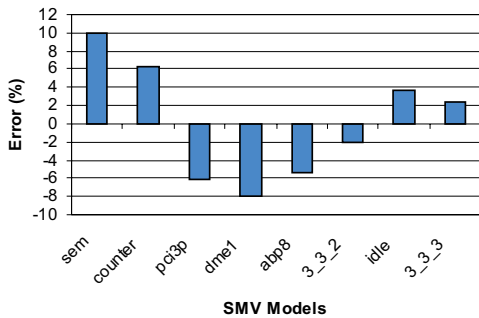


Figure 3. MSP-IA prediction error.

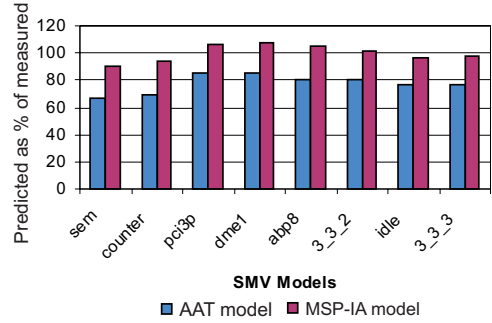


Figure 4. AAT and MSP-IA estimates.

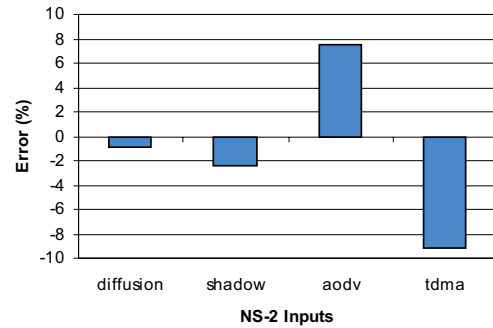


Figure 5. MSP-IA prediction error.

Figure 4 and 6. Hence, for these applications, AAT is a too optimistic approach. Written in C/C++, SMV and NS2 have highly irregular access patterns. Because most of their data accesses can not be overlapped, SMV and NS2 have a lower bandwidth transfer.

5.3. Discussion

The experimental results show that the average access time (AAT) is a too conservative machine characterization for the regular applications and too optimistic for the irregular codes. On the other hand, AAT might be the right machine characterization for mixed codes. However, modeling mixed codes is complicated, as the relative importance of the regular and irregular access performance changes dynamically as the program executes.

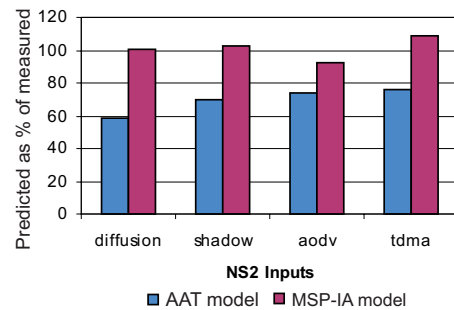


Figure 6. AAT and MSP-IA estimates.

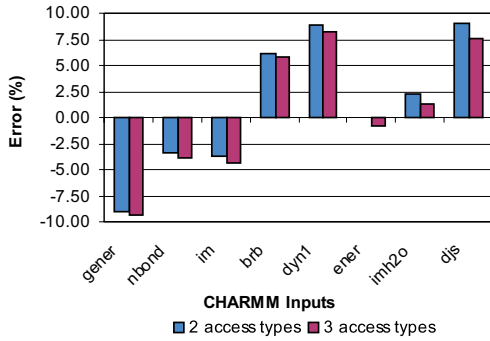


Figure 7. MSP-RA prediction error.

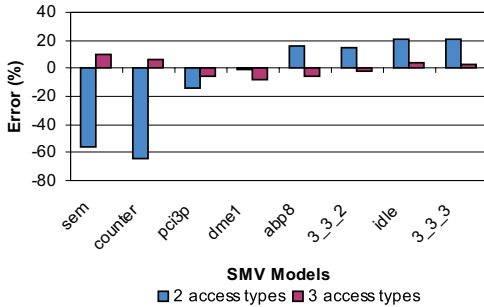


Figure 8. MSP-IA prediction error.

To further investigate whether more complex or simpler models can provide better performance estimates, we do two experiments. First, we extend the MSP-RA model, which models the performance of continuous and strided accesses, to include the performance of accesses within the same cache line. Figure 7 shows that there is no need for a separate performance characterization of these accesses, as the accesses' spatial locality is already captured.

In the second experiment, we simplify the MSP-IA model and consider all non-continuous accesses as being random accesses. The new model considers continuous and random accesses only, and does not model the accesses within the same cache line separately. Not modeling these accesses increases the prediction error considerably, as shown by Figure 8. This precision drop is mainly induced

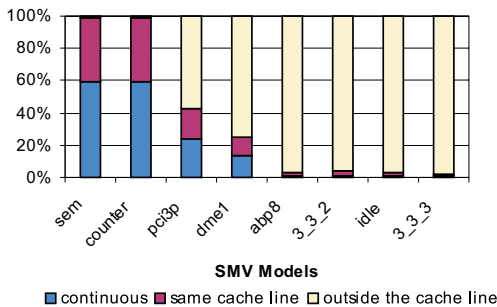


Figure 9. Access type distribution for DRAM reads.

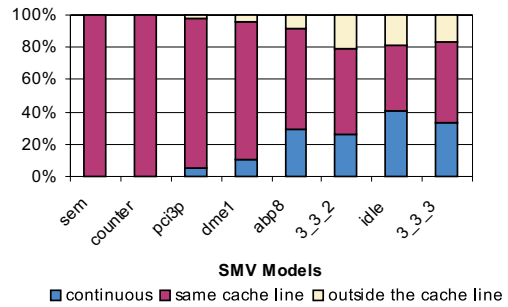


Figure 10. Access type distribution for DRAM writes.

by the fact that the considered SMV models have most of the DRAM writes within the same cache line (see Figure 9 and Figure 10). These accesses have a faster transfer bandwidth than random accesses. Therefore, to accurately estimate the performance of irregular codes, three types of memory access have to be considered.

These experiments show that simplifying the models increases the prediction error considerably, and complicating the models doesn't improve prediction accuracy.

6. Concluding Remarks

We present two performance evaluation approaches that predict the performance of programs depending heavily on the speed of contiguous, strided and random memory access streams. These models compute the execution time of a program as the sum of the times the program spends at each memory level. The time the program spends at a memory level is a linear combination of the number of times each access type appears at that level multiplied by the time to execute that access. The speed of contiguous, strided and random memory accesses can be obtained through measurements on real machines, or through other means for designs that are not yet realized.

The increasing complexity of current and future microprocessors makes performance prediction of real applications difficult. A model that predicts performance of a large class of applications must capture all the system details that have a non-negligible impact on an application's performance yet be simple enough to allow fast evaluation. To allow simple models, we restrict our attention to two classes of programs, regular and irregular programs. Membership in these classes is defined by the memory access pattern of an application (for regular applications, 80+% of their non-contiguous accesses are strided, for irregular applications, 80+% of their non-contiguous accesses are random). Since the memory access pattern classifies the applications, it is not surprising that simple models with few parameters based on memory access properties perform well. Other models, or detailed simulations, are without doubt useful in specific scenarios, but the simple models presented here

allow an evaluation of a system for real applications with large data spaces.

Whereas a machine characterization used together with a characterization of regular codes might want to include a large number of stride values, good results can be obtained from a simplified model that relies only on two data points: continuous and strided (regular) accesses. Furthermore, to accurately predict the execution time of an irregular application (written in C/C++) it is sufficient to characterize machine performance for three groups of access strides: continuous accesses, accesses within a cache line, and random accesses.

The methods require information that is easy to acquire. The *mempref* benchmark gives the different access times of the memory system used. The *shade* tool provides the memory access demands of an application, i.e. the number of times each access type appears at each level of the memory hierarchy. We have successfully applied these prediction models to executions of CHARMM, SMV, and NS2 on a real SPARC system with a three-level memory hierarchy.

These performance models are relatively simple and do not require object code analysis. The errors of their performance estimates are in the range of $\pm 10\%$; the models strive to find a reasonable compromise between the simulation time and the accuracy of the performance estimates they produce. These models are therefore useful for those performance studies that involve complete complex programs with large data sets.

Acknowledgments

We thank Tom Stricker for his comments on earlier drafts of the paper.

References

- [1] S. Kaplan, Y. Smaragdakis, and P. Wilson, "Trace Reduction for Virtual Memory Simulations," in *Proc. SIGMETRICS*, pp. 47–58, 1999.
- [2] Y. Smaragdakis, S. Kaplan, and P. Wilson, "EELRU: Simple and Effective Adaptive Page Replacement," in *Proc. SIGMETRICS*, pp. 122–133, May 1999.
- [3] M. Kjelso, M. Gooch, and S. Jones, "Performance Evaluation of Computer Architectures with Main Memory Data Compression," *Journal of Systems Architecture*, vol. 45, pp. 571–590, 1999.
- [4] P. Wilson, S. Kaplan, and Y. Smaragdakis, "The Case for Compressed Caching in Virtual Memory Systems," in *Proc. 1999 USENIX Tech. Conf.*, pp. 101–116, June 1999.
- [5] R. Uhlig and T. Mudge, "Trace-Driven Memory Simulation: A Survey," *ACM Computing Surveys*, vol. 29, pp. 128–170, June 1997.
- [6] K. Gallivan, D. Gannon, W. Jalby, A. Malony, and H. Wijshoff, "Behavioral Characterization of Multiprocessor Memory Systems: A Case Study," in *Proc. SIGMETRICS*, pp. 79–88, Apr. 1989.
- [7] R. Saavedra and A. Smith, "Measuring Cache and TLB Performance and Their Effect on Benchmark Run Times," *IEEE Trans. Computers*, vol. 44, pp. 1223–1235, Oct. 1995.
- [8] R. Saavedra and A. Smith, "Analysis and Benchmark Characteristics and Benchmark Performance Prediction," *ACM Trans. Computer Systems*, vol. 14, pp. 344–384, Nov. 1996.
- [9] D. Ofelt and J. Hennessy, "Efficient Performance Prediction for Modern Microprocessors," in *Proc. SIGMETRICS*, pp. 229–239, 2000.
- [10] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 3rd ed., 2002.
- [11] A. Ailamaki, D. DeWitt, M. Hill, and D. Wood, "DBMSs on a Modern Processor: Where Does Time Go?," in *Proc. VLDB*, pp. 266–277, Sept. 1999.
- [12] C. Cascaval, L. DeRose, and D. Padua, D. Reed, "Compile-time Based Performance Prediction," in *Proc. LCPC Workshop*, pp. 365–379, Aug. 1999.
- [13] S. Ghosh, M. Martonosi, and S. Malik, "Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity," in *Proc. ASPLOS*, pp. 228–239, Oct. 1998.
- [14] D. Sorin, V. Pai, S. Adve, M. Vernon, and D. Wood, "Analytic Evaluation of Shared-Memory Systems with ILP Processors," in *Proc. ISCA*, pp. 380–391, June 1998.
- [15] R. Jin and G. Agrawal, "Performance Prediction for Random Write Reductions: A Case Study in Modeling Shared Memory Programs," in *Proc. SIGMETRICS*, pp. 117–128, June 2002.
- [16] R. Bryant and D. O'Hallaron, *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2002.
- [17] C. Hristea, D. Lenoski, and J. Keen, "Measuring Memory Hierarchy Performance of Cache-Coherent Multiprocessors Using Micro Benchmarks," in *Proc. ICS*, pp. 1–12, Nov 1997.
- [18] T. Stricker and T. Gross, "Optimizing Memory System Performance for Communication in Parallel Computers," in *Proc. ISCA*, pp. 308–319, June 1995.
- [19] C. Ding and K. Kennedy, "Memory Bandwidth Bottleneck and its Amelioration by a Compiler," in *Proc. IPDPS*, pp. 181–190, 2000.
- [20] L. McVoy and C. Staelin, "Lmbench - Tools for Performance Analysis." <http://www.bitmover.com/lmbench/>.
- [21] J. McCalpin, "STREAM Sustainable Memory Bandwidth in High Perf. Computers." <http://www.cs.virginia.edu/stream>.
- [22] T. Stricker and T. Gross, "Global Address Space, Non-Uniform Bandwidth: A Memory System Performance Characterization of Parallel Systems," in *Proc. HPCA*, pp. 168–180, Feb. 1997.
- [23] T. Stricker and C. Kurmann, "ECT memperf - Extended Copy Transfer Characterization." <http://www.cs.inf.ethz.ch/CoPs/ECT/>.
- [24] T. Conte and C. Gimarç, *Fast Simulation of Computer Architectures*. Kluwer Academic Publishers, 1995.
- [25] "Shade." <http://www.sun.com/microelectronics/shade/>.
- [26] A. Badawy, A. Aggarwal, D. Yeung, and C. Tseng, "Evaluating the Impact of Memory System Performance on Software Prefetching and Locality Optimizations," in *Proc. ICS*, pp. 486–500, June 2001.
- [27] D. Agarwal, W. Liu, and D. Yeung, "Exploiting Application-Level Information to Reduce Memory Bandwidth Consumption," in *Proc. 4th Workshop on Complexity-Effective Design*, June 2003.
- [28] "SMV." <http://www-2.cs.cmu.edu/~bwolen/software/>.
- [29] B. Yang, R. Bryant, D. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. Ranjan, and F. Somenzi, "A Performance Study of BDD-Based Model Checking," in *FMCAD'98*, pp. 255–289, Nov. 1998.
- [30] V. Schuppan and A. Biere, "A Simple Verification of the Tree Identify Protocol with SMV," in *Proc. IEEE 1394 (FireWire) Workshop*, pp. 31–34, March 2001.
- [31] B. Brooks, R. Bruccoleri, B. Olafson, D. States, S. Swaminathan, and M. Karplus, "CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations," *Comp. Chem.*, no. 4, pp. 87–217, 1983.
- [32] "NS-2 Network Simulator." <http://www.isi.edu/nsnam/ns/>.