CMP$im: A Pin-Based On-The-Fly Multi-Core Cache Simulator

Aamer Jaleel†, Robert S. Cohn†, Chi-Keung Luk†, Bruce Jacob‡

†Intel Corporation, VSSAD
‡University of Maryland, College Park
{aamer.jaleel@intel.com}

Abstract

Chip multiprocessors are the next attractive point in the design space of future high performance processors. There is a growing need for simulation methodologies to determine the memory system requirements of emerging workloads in a reasonable amount of time. To explore the design space of a CMP memory hierarchy, this paper presents the use of binary instrumentation as an alternative to execution-driven and trace-driven simulation methodologies. Using the binary instrumentation tool, Pin, we present CMP$im to characterize cache performance of single-threaded, multi-threaded, and multi-programmed workloads at the speeds of 4-10 MIPS. For memory intensive single-threaded workloads, the cache performance reported by CMP$im is three orders of magnitude faster and within 4% of an cycle-accurate x86 performance model.

1. Introduction

Recent industry trends show that the future of high performance computing will be defined by the performance of multi-core processors [1, 2, 3]. As a result, processor architects now face key design decisions in designing the memory hierarchy. Additionally, as parallel applications become common workloads that execute on CMPs, detailed memory characteristics of these emerging workloads are essential in designing an efficient memory hierarchy. Such characterization and exploratory studies require fast simulation techniques that can compare and contrast the performance of alternative design policies. This paper demonstrates the use of binary instrumentation tools as an alternative to existing execution-driven and trace-driven methodologies. Using the binary instrumentation tool Pin, we present CMP$im to characterize the memory system performance of workloads on multi-core processors.

Simulation is a common methodology that is used both for design space exploration and the identification of performance bottlenecks in existing systems. There exist many free simulators and software tools to investigate the memory system performance of applications. In general, memory system simulators fall into two main categories: trace-driven or execution-driven [24]. With trace-driven cache simulators, pre-collected address traces are used to feed a cache simulator (e.g. Dinero IV [11]). Such simulators rely on existing tools to collect an applications memory address trace and log them to file for later use. Execution-driven cache simulators rely on functional/performance models to execute an application binary. The memory addresses generated by the functional/ performance model are fed, in real time, to a cache simulator modeled within the functional/performance model. Among the two, trace-driven simulation is a popular technique for conducting memory performance studies [24].

The usefulness of trace-driven simulation, however, lies in the continued availability of memory address traces to study the memory performance of different workloads. With several emerging application domains, understanding the memory behavior and cache requirements of different applications requires the ability to generate address traces by just about anyone. Address trace generation for a target ISA can require sophisticated hardware tools [24] or functional models that not only support the target ISA but also the requirements of the workload. For example, the functional model must provide support for multiple contexts if executing a multi-threaded workload. Such infrastructure to capture memory address traces can be expensive and/or complex to build.

Even if address trace generation were trivial, a practical problem with storing memory address traces is that the address traces can be large, potentially occupying several gigabytes of disk space even in their compressed formats. Consequently, transferring and sharing large address traces between different locations can be inconvenient. Furthermore, another problem of using address traces is that the trace is only representative of the compiler and compiler optimizations used to compile the workload. As a result, studying the behavior of different compiler optimizations of the workload requires the creation of address traces for each compiler and compiler optimization type. Ideally, a desirable approach for conducting memory performance studies is to have the benefits of the execution-driven methodology without incurring the associated slow speeds and complexities.

To address the drawbacks of current methodologies, the main contribution of this paper is to illustrate the use of existing binary instrumentation tools to conduct quick exploratory memory performance studies. We present, CMP$im, a memory system simulator that uses the Pin [4, 16] binary instrumentation system. With Pin serving as the

functional model that provides CMP$im with memory addresses, CMP$im processes memory requests generated by a workload on the fly i.e. in real time. Thus, CMP$im simulates the memory system performance of a workload without the overhead of dealing with large address trace files.

CMP$im is fully configurable and can gather detailed cache performance statistics as well as the total amount of sharing between different threads of a multi-threaded application. Specifically, CMP$im users can vary the cache parameters, allocation/replacement policies, and write policies. Users can also specify the number of levels in the cache hierarchy with specifications on the type of inclusion policy. Even more, users can configure some or all levels of the cache hierarchy to be shared or private amongst different threads/cores of the simulated CMP. Our studies show that CMP$im provides detailed cache statistics over billions of instructions of an application (or regions of application specified by the user) at the rate of 4-10 million instructions per second (MIPS).

Unlike existing simulation techniques, the main advantages of CMP$im are: it is a *parallel* model that can process memory requests from multiple threads at the same time; it is *fast*; it is *flexible*-users can model any kind of cache hierarchy; it can model multi-cores and multi-threaded cores; it can easily run complex applications like Oracle and Java without any user support; and finally it is relatively *simple* when compared to full performance models, thus, making it easy to extend or modify.

For several SPEC CPU2006 workloads we correlated the last-level cache (LLC) performance of CMP$im with a detailed cycle accurate x86 performance model. Our results show that the miss rate reported by CMP$im is within 13% of the cycle accurate performance model. For applications with more than one MPKI (miss per thousand instructions), the average MPKI is within 4%. For the regions simulated, CMP$im is 100-1000x faster than the speed of the cycle accurate performance model. The results are appealing as CMP$im enables quick exploratory studies with reasonable correlation to a detailed timing model.

To showcase the use of CMP$im, we present a full-run memory characterization of representative SPEC CPU2006 workloads, a multi-threaded workload *ammp* from the SPECOMP benchmark suite [5], and finally a 8-core multi-programmed workload mix of SPEC CPU2006 workloads.

2. Background

2.1. Pin - A Binary Instrumentation Tool

Pin [4, 18] is a dynamic binary instrumentation system for Linux and Windows binaries running on Intel® IA-32 (x86 32-bit), IA-32E (x86 64-bit), and Itanium® processors. Pin is similar to the ATOM toolkit [22] and provides infrastructure for writing program analysis tools called Pin tools.

The two main components of a Pin tool are: *instrumentation* and *analysis* routines. Instrumentation routines utilize the rich API provided by Pin to insert calls to user defined analysis routines. These calls are inserted by the user at arbitrary points in the application instruction stream. Instrumentation routines define the characteristics of an application to instrument. Analysis routines are called by the instrumentation routines at application run time. For example, using the Pin API [4], a user can write an instrumentation routine to instrument every instruction executed by an application. If the instrumentation routine sets up a call to a user defined analysis routine DoCount() (which simply increments a counter), then the Pin tool counts the total number of dynamic instructions executed by the program. Besides writing such simple utilities, Pin provides many other advanced features to conduct a variety of micro architecture studies. For example, customized Pin tools can profile the static or dynamic distribution of instructions executed by a given application, determine the outcomes of branch instructions and their associated branch targets, acquire effective addresses of all memory instructions executed, change architectural state of registers [22]. With such information, users can write customized Pin tools that model branch predictors, cache simulators, and simple performance models.

Besides instrumenting single-threaded applications, Pin also supports the instrumentation of multi-threaded applications. The scheduling of different threads of the application is controlled by the operating system. To distinguish between the different threads of the application, Pin assigns each thread with a unique ID which is different from the native process ID assigned by the operating system. Pin assigns the first thread, i.e. the main thread, with thread ID 0 and each additional new thread is assigned the next sequential ID, i.e. 1, 2, 3, and so on. Thus, when conducting studies with a four-threaded workload, Pin distinguishes between threads by assigning the main thread with thread ID 0, and the three remaining threads with thread IDs 1, 2 and 3. It is the responsibility of the Pin tool to distinguish instrumentation based on different thread IDs.

2.2. Related Work

Several studies have used trace-driven or execution-driven methodologies to characterize the memory behavior and performance of different types of applications. Uhlig et al. [24] provide a detailed survey of existing trace-driven methodologies. Iyer et al. [13] introduced a trace-driven simulation framework called CASPER to explore different cache organization alternatives, prefetching mechanisms, coherence protocols and other research studies. Jaleel et al. [14, 15] used Pin to conduct a detailed memory characterization study of parallel workloads on CMPs. Nurvitadhi et al. [20] used an FPGA based cache model

(PHA$E) that connects directly to the front-side bus to understand the L3 cache behavior of SPECjAppServer and TPC-C. Abandah et al. [7, 8] proposed a configuration independent approach to analyze the working set, concurrency, communication patterns, as well as sharing behavior of shared memory applications. They present a tracing tool called Shared-Memory Application Instrumentation Tool (SMAIT) to measure different sharing characteristics of the NAS shared-memory applications. Barroso et al. [9] characterized the memory system behavior of commercial workloads such as Oracle, TPC-B, TPC-D, and AltaVista search engine. They did their characterization of the memory system behavior using ATOM [23], performance counters on an Alpha 21164 as well as the SimOS simulation environment. Woo et al. [25] characterized several aspects of the SPLASH-2 benchmark suite. They used an execution-driven simulation with the Tango Lite tracing tool [12]. Perl et al. [21] studied Windows NT applications on Alpha PCs and characterized application bandwidth requirements, memory access patterns, and application sensitivity to cache size. Chodneker et al. [10] analyzed the time distribution and locality of communication events in some message-passing and shared-memory applications.

The work presented in this paper differs from prior work in that it presents binary instrumentation as an alternative approach to study cache performance of workloads. We introduce CMP$im, a CMP cache simulator that can characterize the memory behavior of single and multi-threaded workloads. We believe that CMP$im also fills the gap on the lack of simple x86 performance tools to characterize the memory behavior of applications across different memory system configurations. Full system simulators such as Bochs [17] and Simics [19] support the x86 ISA, however they emulate an entire system with peripherals and an operating system. Even though such tools are valuable for research, characterizing the memory behavior of individual workloads on such systems can be non-trivial and rather slow. Unlike existing simulators, CMP$im can characterize the behavior of applications over their entire run or periods of interest defined by the user without using tracing mechanisms, performance counters, or bus sniffers. Existing work has demonstrated the usefulness of Pin to conduct performance analysis of different applications. Reddi et al. [22] discuss the use of Pin as a tool for computer architecture research and education. We demonstrate a working example of CMP$im based on Pin that can be used to characterize application behavior and memory performance of emerging x86 workloads on CMPs.

3. CMP$im - A Multi-Core Cache Simulator

The interfaces to most binary instrumentation tools are API calls that allow users to insert instrumentation and analysis routines. The instrumentation routine defines WHERE in the application to insert instrumentation while the analysis routine defines WHAT to do when the instrumentation is activated.

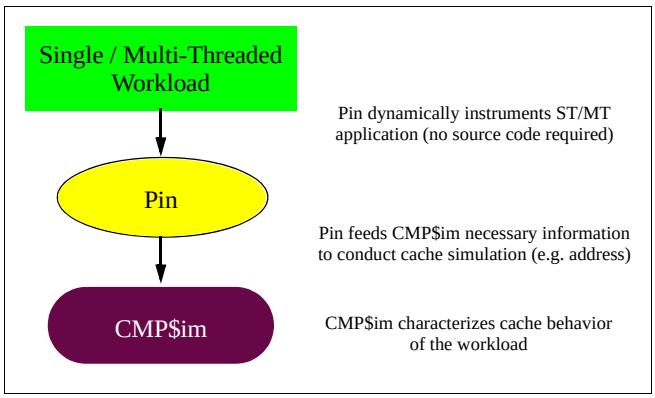

Figure 1: CMP$im Implementation Overview.

To simulate the cache behavior of an application, all memory references generated by a workload must be captured and played through a cache model. The Pin API is used to capture the type of memory operation (read/write), the memory address, the size of the memory operation (in bytes), and finally the Thread ID of memory reference (if a multi-threaded application is being studied).

Figure 1 graphically illustrates the interaction between workloads, Pin, and CMP$im. The user application runs on top of Pin and Pin provides to CMP$im the necessary memory instruction information. The memory instruction information extracted by the Pin API is dynamically sent to a cache model that handles user defined cache sizes, associativity, and allocation and replacement policies. The cache hierarchy is fully configurable where the user can specify the number of levels in the cache hierarchy, whether the levels are shared or private, and the appropriate inclusion policy. CMP$im also supports an invalidate-based cache coherence protocol (similar to MESI). Finally, since Pin supports instrumentation of multi-threaded workloads, it was required that CMP$im be thread safe. As a result, CMP$im is a parallel software model that makes use of shared memory primitives (e.g. locks) to guarantee correct behavior.

3.1. Multi-Programmed Workload Simulation

Since multi-core processors enable concurrent execution of multiple programs, it is also highly desirable for simulation techniques that enable quick multi-programmed memory system simulation.

Pin-based multi-programmed CMP$im simulation would require a single instance of Pin to simultaneously instrument multiple applications. However, currently Pin only supports instrumenting a single application. Since CMP$im already supports multi-threaded applications, we extended CMP$im to enable multi-programmed simulation. Instead of rewriting Pin to support multiple programs at the same time, the cache

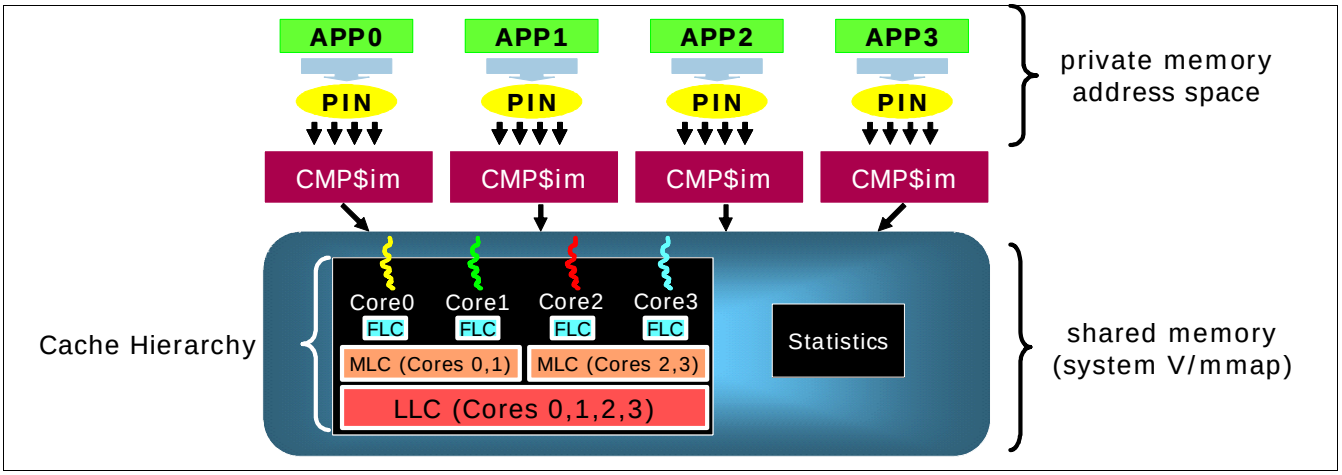

Figure 2: Multi-Programmed CMP$im Implementation Overview.

hierarchy was created in shared memory (using System V or memory mapped I/O) and required multiple instances of Pin and CMP$im to connect to the shared memory (see Figure 2). We distinguished identical virtual addresses between the different applications by comparing the application ID along with the virtual address. Once the required number of applications connects to the shared memory, cache simulation proceeds normally.

3.2. CMP$im Statistics

CMP$im gathers statistics such as the total number of cache accesses and misses, sharing characteristics of multi-threaded applications, coherence traffic, and much more. All statistics are output to a data file when the program finishes execution. Alternatively, to characterize the time varying behavior of the application, statistics can also be logged periodically to the output file. This enables users to visualize the time varying behavior of an application over the course of simulation and helps identify representative regions of execution for detailed simulation. A detailed listing of statistics collected by CMP$im is provided in [15].

4. Experimental Methodology

For our studies, we assume a multi-core system with one thread per-core. We model a three level cache hierarchy. The L1 and L2 caches are private to each core and the L3 cache is either configured to be private or shared. We used CMP$im to characterize the run time memory behavior of the SPEC CPU2006 suite on the reference input sets, the *ammp* workload from the SPECOMP [5] suite with the reference input set, and a multi-programmed mix of 8 SPEC CPU2006 workloads. The L1 data cache is 32KB, 2-way set associative, with 64B line size. The L2 cache is 256KB, 8-way set associative, with 64B line size. The L3 cache is 16-way set associative, with 64B line size and write-back policy. All caches allocate on a store miss and use the LRU replacement policy. A MESI cache coherence protocol is also modeled.

All workloads are run on a system of Intel® Pentium® 4 3.2 GHz processors and are compiled using the icc compiler, with optimization flags -O3. Except for the multi-programmed workload mix, all workloads were run to completion with statistics logged to file every 10 million instructions. For the 8-core workload mix, simulations were run until the last application executed a billion instructions. After simulation, the behavior of the workload over the different intervals of execution is depicted graphically.

5. Results

We showcase the use of CMP$im by presenting the cache performance studies of single-threaded, multi-threaded, and multi-programmed workloads.

5.1. Single-Threaded Workloads

Figure 3a presents the last-level cache (LLC) performance of representative SPEC CPU2006 workloads run to completion on a three-level cache hierarchy with a 2MB LLC. The x-axis represents the total number of instructions executed in billions and the y-axis represents the miss rate of the application. The time varying behavior of the workload is presented at a ten million instruction granularity (represented by the green-dotted lines). The cumulative behavior of the workload is represented by the red solid line. Based on our studies, CMP$im can characterize different phases of execution of the single-threaded workloads (e.g. for *tonto* over 3 trillion instructions) at the speed of 8-12 MIPS. A detailed memory characterization of all SPEC CPU2000 and CPU2006 workloads run to completion is available at [6].

Figure 3b compares the cache performance results provided by CMP$im to a detailed x86 cycle accurate performance model (on the secondary/right axis). Since CMP$im does not model a prefetcher, we disable the

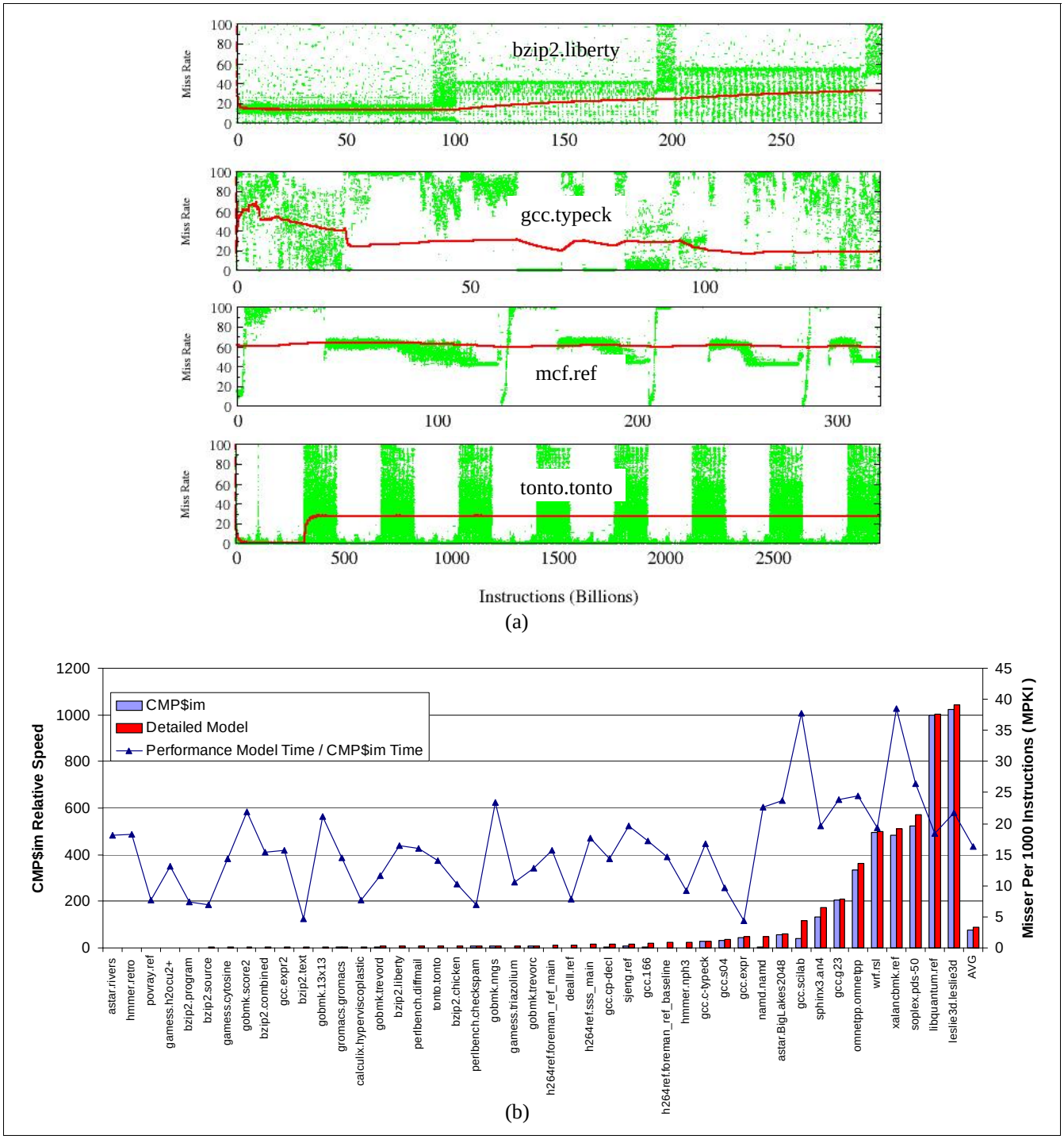

Figure 3: Full Run Cache Characterization of ST Workloads and Comparison of CMP$im to Detailed Performance Models.

prefetcher in the detailed performance model. Both CMP$im and the cycle accurate performance model ran identical regions of the SPEC CPU2006 benchmark suite. The metric of comparison is misses per 1000 instructions (MPKI). The results show CMP$im is within 13% of the cycle accurate model. For applications with more than one MPKI, CMP$im is within 4% of the cycle accurate model. The difference between CMP$im and the detailed performance model are most likely due to the lack of speculation and out-of-order exeuction in CMP$im.

Figure 3b also shows the relative speed of CMP$im compared to the speed of the cycle accurate performance model. The results show that CMP$im is two to three orders of magnitude (average 435x) faster than the cycle accurate

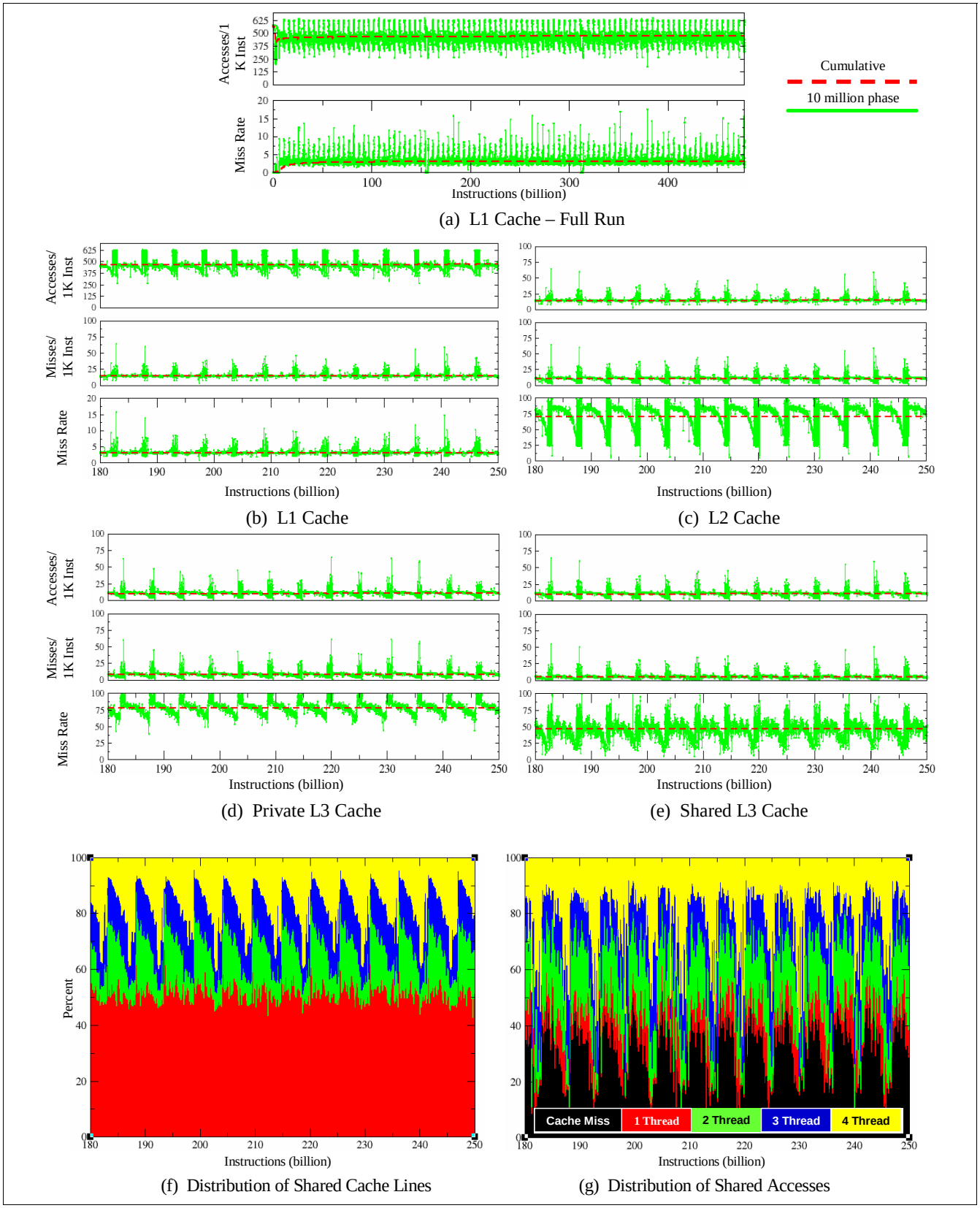

Figure 4: Cache Performance and Sharing Characteristics of a Multi-Threaded Workload using CMP$im.

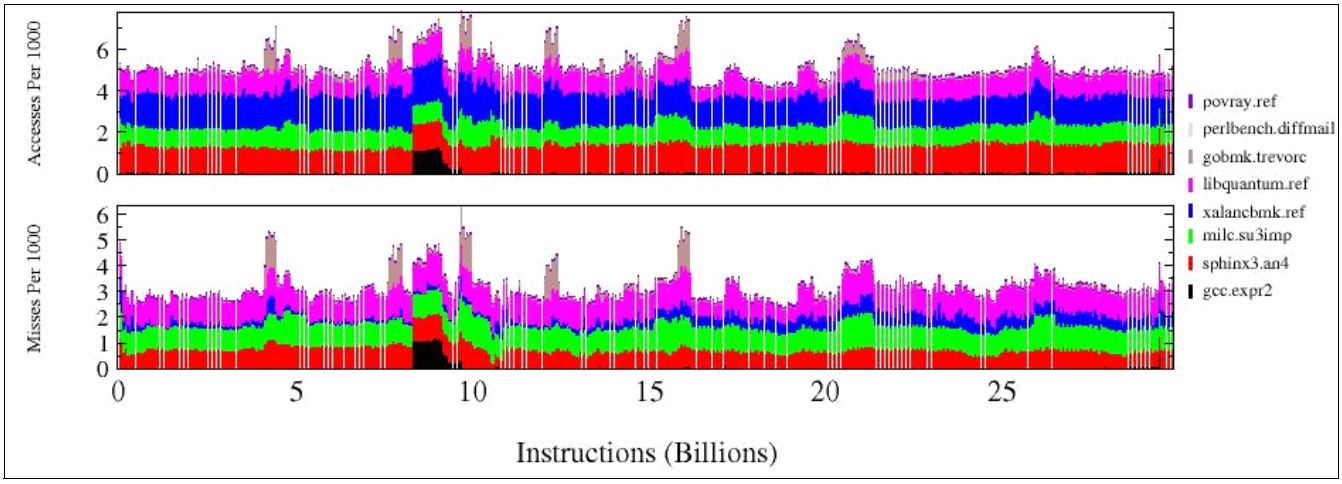

Figure 5: Cache Performance Study of an 8-core CMP sharing the Last-Level Cache.

performance model. These results are appealing as CMP$im enables quick exploratory studies with reasonable correlation to a cycle accurate detailed performance model.

5.2. Multi-Threaded Workloads

We now present the cache performance of the SPECOMP workload *ammp*. The workload was run to completion using CMP$im at a speed of 5 MIPS. Figure 4a shows the number of L1 cache accesses and misses of *ammp* when run to completion. The x-axis represents the total number of instructions (in billion) and the y-axis represents the accesses per 1000 instructions and miss rate of the L1 cache. The figure shows that the workload has a periodic pattern with approximately 90 loops. Since the workload exhibits a periodic behavior, we arbitrarily zoomed in on the region between 180 and 250 billion instructions.

Figures 4b, 4c, 4d, and 4e illustrate the L1, L2, private L3, and shared L3 cache behavior for the selected region of execution. For each cache, we present the total number of cache accesses per 1000 instructions, the total number of cache misses per 1000 instructions, and the miss-rate. The figure shows a periodic behavior in the cache access and miss pattern with one period spanning 4 billion instructions. Each loop begins with a cache miss rate as high as 95%. The cache miss rate reduces to 40% during the second half of the loop because of improved locality. Despite the existence of locality, the large miss rates in the L2 cache imply a working set larger than the size of the L2 cache (256KB).

Figures 4d and 4e shows the L3 cache behavior of *ammp* for private and shared cache configuration. The size of the L3 cache in the shared configuration is 2MB and the size of each L3 cache in the private configuration is 512KB. On average, the shared configuration has 25% fewer misses than a private configuration. This is because the larger effective capacity of a shared cache allows it to accommodate to the variable working set of each core. Additionally, if multiple cores share a cache line, a shared cache reduces miss rate by making full use of the cache capacity by avoiding duplication.

Figure 4f presents the distribution of cache lines shared between different cores of the CMP. The x-axis represents the total number of instructions and the y-axis represents the distribution of cache lines that are either private, or shared between two, three, or four cores. The bottom-most segment represents private cache lines, followed by cache lines shared by two cores, three and four cores respectively. Figure 4f shows that half the cache is shared by two or more cores and Figure 4g shows as much as 50-80% of the last-level cache accesses are to lines that are shared by two or more cores. With the significant amount of data sharing between multiple cores, *ammp* performs better with a shared last-level cache.

5.3. Multi-Programmed Workloads

We finally present the use of multi-programmed CMP$im to study the cache performance behavior of eight SPEC CPU2006 workloads sharing a 16MB LLC. Figure 5 shows the time varying behavior of the workloads with the first graph showing access per 1000 instructions and the second graph showing misses per 1000 instructions. The x-axis presents the total number of instructions executed by all applications. For each metric, we present the distribution of references and misses for each workload in the mix. From the figure, we observe that of the eight workloads, four of the workloads have significant activity in the shared LLC. We also observe applications going through different phases of execution which result in an increase or decrease in cache misses. For example, at about 8 billion instructions we observe *gcc.expr2* change phases and suffer an increase in cache accesses and misses. The characterization of the multi-programmed workload mix occurred at approximately 5 MIPS.

6. Conclusions

This paper illustrates the use of binary instrumentation as an alternative to execution-driven and trace-driven methodologies. Using the binary instrumentation system Pin, we present a memory system simulator, CMP$im, that is fast, flexible, easy to use, and simple to modify. We demonstrated the use of CMP$im to characterize the memory system performance of single-threaded and multi-threaded workloads run to completion. We also demonstrated the use of CMP$im to conduct multi-programmed workload simulation using shared memory programming.

Since binary instrumentation using Pin normally occurs at the speed of native execution, we show that CMP$im-driven cache simulation allows for full run characterization of workloads at speeds ranging from 4-10 MIPS. Compared to a detailed cycle accurate performance model, CMP$im is two to three orders of magnitude faster. Correlating CMP$im with the detailed performance model showed that the cache performance reported by CMP$im is within 17% (4% for memory bound applications) of a detailed performance model.

As part of our on-going work, we are correlating multi-threaded and multi-programmed cache performance with the cycle accurate performance models. We are also investigating techniques to incorporate a performance model within the CMP$im infrastructure without sacrificing speed. For example we are exploring the use of analytical performance models as proposed by [16].

Besides cache performance studies, CMP$im can also be used to explore the design space of TLBs and prefetching. CMP$im can be used to investigate novel cache replacement and advanced cache management studies such as providing quality of service guarantees when executing multi-programmed workloads on CMPs.

References

[1] AMD MultiCore Technology: http://multicore.amd.com

[2] IBM Cell Processor: http://www-306.ibm.com/chips/techlib/techlib.nsf/products/Cell

[3] Intel Pentium Processor Extreme Edition: http://www.intel.com/products/processor/pentiumXE/index.htm

[4] Pin home page: http://rogue.colorado.edu/Pin/.

[5] SPECOMP2001 http://www.spec.org/

[6] SPEC CPU2000 and SPEC CPU2006 Cache Performance Analysis at http://www.glue.umd.edu/~ajaleel/workload/.

[7] G. A. Abandah and E. S. Davidson. “Configuration Independent Analysis for Characterizing Shared-Memory Applications.” In Proceedings of the 12th. International Parallel Processing Symposium (IPPS), Orlando, Florida, 1998.

[8] G. A. Abandah. “Characterizing Shared-Memory Applications: A Case Study of the NAS Parallel Benchmarks.” Technical Report, HPL-97-24, Hewlett Packard.

[9] L. A. Barroso, K. Gharachorloo, and E. Bugnion. “Memory System Characterization of Commercial Workloads.” In Proceedings of the 25th International Symposium on Computer Architecture (ISCA), Barcelona, Spain, 1998.

[10] S. Chodnekar, V. Srinivasan, A. Vaidya, A. Sivasubramaniam, and C. Das. “Towards a Communication Characterization Methodology for Parallel Applications.” In Proceedings of the International Conference on High Performance Computer Architecture (HPCA), San Antonio, Texas, 1997.

[11] J. Edler and M. D. Hill. “Dinero IV Trace-Driven Uniprocessor Cache Simulator”.

[12] S. Goldschmidt and J. Hennessey. “The Accuracy of Trace-Driven Simulations of Multiprocessors.” Tech Rep. CSL-TR-92-546, Stanford University, Sept. 1992.

[13] R. Iyer. “On Modeling and Analyzing Cache Hierarchies using CASPER.” In Proceedings of the 11th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2003.

[14] A. Jaleel, M. Mattina, and B. Jacob. “Last Level Cache (LLC) Behavior of Data-Mining Workloads on a CMP - A Case Study of Parallel Bioinformatics Workloads.” In Proceedings of the 12th International Symposium on High Performance Computer Architecture (HPCA), Austin, Texas, 2006.

[15] A. Jaleel, R. Cohn, C. K. Luk, and B. Jacob. “CMP$im: A Binary Instrumentation Approach to Modeling Memory Behavior of Workloads on CMPs.” Technical Report - UMD-SCA-2006-01

[16] T. Karkhanis and J. E. Smith. “A First-Order Superscalar Processor Model.” In Proceedings of the 31st International Symposium on Computer Architecture (ISCA), Munich, Germany, 2004.

[17] K. Lawton. Bochs. http://bochs.sourceforge.net.

[18] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation.” In Proceedings of Programming Language Design and Implementation (PLDI), Chicago, Illinois, 2005.

[19] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, and G. Hallberg. Simics: A full system simulation platform. IEEE Computer, 35(2):50-58, Feb. 2002.

[20] E. Nurvitadhi, N. Chalainanont, and S. L. Lu. “Characterization of L3 Cache Behavior of

SPECjAppServer2002 and TPC-C." In Proceedings of the 19th International Conference on Supercomputing (ICS), Boston, Massachusetts, 2005.

[21] S. E. Perl and R. L. Sites. "Studies of Windows NT performance using dynamic execution traces." In Proceedings of the 2nd International Symposium on Operation Systems Design and Implementation (OSDI), Seattle, Washington, 1996.

[22] V. Reddi, A. M.Settle, D. A. Connors and R. S. Cohn. "Pin: A Binary Instrumentation Tool for Computer Architecture Research and Education." In Proceedings of the Workshop on Computer Architecture Education, June 2004.

[23] Srivastava and A. Eustace. "ATOM: A System for Building Customized Program Analysis Tools", Programming Language Design and Implementation (PLDI), 1994, pp. 196-205.

[24] R. A. Uhlig. And T. N. Mudge. "Trace-driven Memory Simulation: A Survey", In ACM Computing Surveys, Vol. 29, 1997.

[25] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. "The SPLASH-2 Programs: Characterization and Methodology Considerations." In Proceedings of the 22nd International Symposium on Computer Architecture (ISCA), Santa Margherita Ligure, Italy, 1995.