CMP$im: A Binary Instrumentation Approach to Modeling Memory Behavior of Workloads on CMPs

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

[†]Intel Corporation, VSSAD

aamer.jaleel@intel.com

[‡]University of Maryland, College Park

blj@eng.umd.edu

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, and data sharing behavior of multi-threaded workloads at the speeds of 4-10 MIPS.

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 application instruction profile and memory system performance of emerging workloads on future 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 [23]. With trace-driven cache simulators, address traces of an application are read from an address trace file and fed to 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. Since functional models of modern ISAs are slow and can be complex to build, trace-driven simulation is a popular technique for conducting memory performance studies [23].

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 working set 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 [23] or functional models that not only support the target ISA but also the requirements of the workload (e.g. 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 memory performance studies. We introduce, CMP$im, a memory system simulator based on the binary instrumentation tool called Pin [4, 16]. With Pin serving as the functional model that provides CMP$im with memory addresses, CMP$im can process memory requests generated by a workload in real time. Thus, CMP$im simulates the memory system performance of a workload without the overhead of dealing with large address trace files. Furthermore, CMP$im can simulate the memory system performance of a multi-threaded workload on a CMP with single or multi-threaded cores. For example, when simulating a four-threaded workload, CMP$im can model a four-core CMP with one thread per core, a two-core CMP with two threads per core, or a single-core processor with four threads per core.

CMP$im is fully configurable and can gather detailed application statistics on cache performance as well as the total amount of data shared between different threads of an application. Specifically, a user can set the cache parameters, allocation/replacement policies, and write policies. A user can also specify the number of levels in the cache hierarchy with specifications on the type of inclusion policy. Even more, the user can configure some levels of the cache hierarchy to either be shared or private amongst different threads/cores of the simulated CMP. CMP$im also provides application instruction profiles, detailed cache statistics on cache reuse, number of cache hits and misses, number of conflict misses, and number of writebacks. CMP$im can gather these statistics for 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* memory system simulator 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.

We used CMP$im to characterize the memory behavior of the multi-threaded workload *ammp* from the SPECOMP benchmark suite [5]. We present the per-thread instruction profile and the time varying cache and sharing behavior of ammp when run to completion. The detailed characterization of ammp reveals that it exhibits mostly read data sharing with 30% of its shared data referenced 50-80% of the time. Additionally, since most of the memory references are to the shared data, ammp benefits greatly from a shared last-level cache configuration compared to a private last-level cache configuration.

BACKGROUND

PIN – A BINARY INSTRUMENTATION TOOL

Pin is a dynamic binary instrumentation tool that instruments an application that executes on Intel processors. Pin supports the instrumentation of Linux, MacOS, and Windows binary executables (without the need for source code) for Intel® Xscale®, IA-32 (x86 32-bit), IA-32E (x86 64-bit), and Itanium® processors. Pin is similar to the ATOM toolkit for Compaq's Tru64 Unix on Alpha processors [22]. Like ATOM, Pin provides an 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 microarchitecture 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 [20]. 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.

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. [23] 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] used Pin to conduct a detailed memory characterization study of parallel data-mining bioinformatics workloads on CMPs. Nurvitadhi et al. [18] 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. [6, 7] 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 [22], performance counters on an Alpha 21164 as well as the SimOS simulation environment. Woo et al. [24] 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. [19] 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 [15] and Simics [17] 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$im can characterize the behavior of applications over their entire run or periods of interest defined by the user without using tracing mechanisms, performance counters, or bus sniffers. Existing work has demonstrated the usefulness of Pin to conduct performance analysis of different applications. Reddi et al. [20] discuss the use of Pin as a tool for computer architecture research and education. We demonstrate a working example of CMP$im based on Pin that can be used to characterize application behavior and memory performance of emerging x86 workloads on uni-processors or CMPs.

CMP$im — An Instrumentation Based Simulator

The interfaces to most binary instrumentation tools are API calls that allow users to hook in their instrumentation routines. In Pin, the API call to *INS_AddInstrumentationFunction()* allows for a user to instrument programs based on a single instruction while the *TRACE_AddInstrumentationFunction()* allows the user to instrument programs at a trace (multiple basic blocks) level. CMP$im uses these two API routines to set up calls to instrumentation routines *Instruction()* and *Trace()*. These two instrumentation routines in turn call the two main analysis routines *MemReference()* and *IProfile()* which are responsible for cache simulation and instruction profiling respectively. We now provide a detailed description of the instrumentation and analysis routines responsible for cache simulation and instruction profiling. Figure 1 illustrates an implementation overview of the CMP$im simulator.

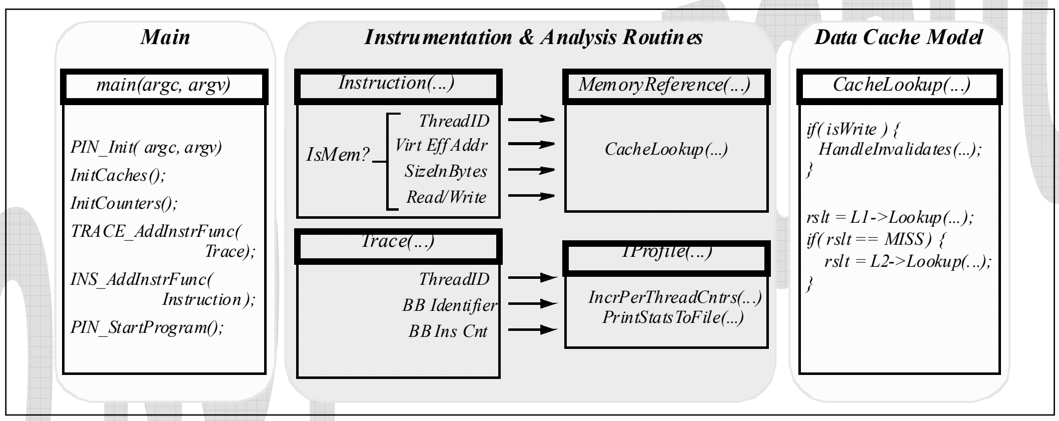

Figure 1: Implementation Overview of CMP$im

Cache Simulation

To simulate the cache behavior of a workload, all memory references generated by a workload must be captured and played through a cache model. In CMP$im, the *Instruction()* instrumentation routine sets up the call to the *MemReference()* analysis routine every time an instruction that references memory is executed. The primary function of the *MemReference()* analysis routine is to simulate the cache behavior of the application based on user defined cache parameters. *MemReference()* takes as input the thread ID of the instruction, the type of memory operation (read/write), the memory address, and finally the size of the memory operation (in bytes). This analysis routine interfaces with a cache model that handles user defined cache sizes, associativity, and allocation and replacement policies. The user has the option of specifying the number of levels in the cache hierarchy. The user can also specify the caches in the hierarchy to be write-through or write-back and the levels of the hierarchy to be inclusive or non-inclusive. Furthermore, the cache model can simulate some levels in the cache hierarchy to be private or shared amongst different threads/cores of a CMP. This functionality allows for a user to simulate a CMP with cores that are single-threaded or multi-threaded. Finally, if multi-threaded workloads are simulated with CMP$im, an invalidate-based cache coherence protocol (similar to MESI) is also implemented. On a write request, invalidates are sent to all relevant private caches to invalidate any matching tag entries. Similarly, on read requests missing in the private cache, remote dirty lines (if any) are required to perform a write-back before servicing the miss in the private cache.

When writing parallel programs, common software programming practice requires that data structures that are shared in a read/write fashion by more than one software context be guarded by a lock to ensure correct behavior. For example, if CMP$im modeled a shared cache without the use of locks, incorrect behavior can occur if two threads simultaneously miss in the cache with different memory addresses that map to the same set in the cache. If the shared cache was not guarded by a lock, both threads can read the same LRU state and fill into the same LRU entry (defeating the purpose of the LRU). Such problems can be avoided by guarding accesses to shared caches by the use of locks. However, a lock per cache can slow the speed of simulation especially if multiple threads contend for the same cache. This source of performance bottleneck was avoided by using a lock per cache set rather than a lock per cache. Thus, during simulation, simultaneous accesses to different sets of the same cache can occur concurrently while simultaneous accesses to the same set of a cache are serialized.

Instruction Profiling

An important aspect of workload characterization is the dynamic distribution of the instruction/opcode mix in an application. This provides insight on whether an application is dependent on memory behavior, compute behavior, or control flow behavior. To determine the instruction profile of an application, CMP$im uses the *Trace()* instrumentation routine to track all basic blocks executed by a program. Instrumenting an application on a basic block basis rather than calculating the instruction profile on a per-instruction basis reduces the total instrumentation overhead. For each basic block, the *Trace()* routine classifies instructions within the basic block into several different categories based on the specifics of Pin's internal x86 instruction decoder XED. For example, instructions are categorized into control instructions, memory read instructions, memory write instructions, integer instructions, and floating point instructions. The *Trace()* routine then sets up a call to the *IProfile()* analysis routine. The arguments passed are: thread ID, pointer to an internal data structure that holds profile information relevant to the basic block, and the instruction count of the basic block. Upon receipt of the different inputs, the *IProfile()* analysis routine increments per thread counters to track the total number of instructions executed for each thread.

In addition to doing instruction profiling, the *Trace()* instrumentation routine is also responsible for setting up calls to the *MemRef()* routine to conduct instruction cache simulation. The *Trace()* routine passes to the *MemRef()* routine the thread ID, the PC of the first instruction in the basic block, and the size of the basic block in bytes. Similar to instruction profiling, instruction cache simulation is done on a basic block basis (rather than per-instruction basis) to reduce the instrumentation overhead.

CHARACTERIZING USER DEFINED REGIONS OF SIMULATION

Besides characterizing the memory behavior and instruction profile of an application over its entire run, CMP$im can also instrument user defined regions of code. If the source code of the application is available to the user, CMP$im provides an API (in the form of a header file) to the user to define start and stop regions into the source code. Specifically, a user can insert a function call to *PINStartInstrumentation()* at the start of the region(s) of interest and a call to *PINEndInstrumentation()* at the end of the region(s) of interest. With this feature, CMP$im dynamically detects user defined start and stop regions and will only instrument applications during the user defined regions of code. However, if source code is unavailable, Pin's API provides mechanisms to skip user-defined instructions or instrument only user-defined functions.

GATHERING STATISTICS WITH CMP$IM

CMP$im can gather a variety of statistics for an application. CMP$im tracks the application instruction profile, the total number of cache accesses and misses, the sharing characteristics of multi-threaded applications, and coherence traffic. These statistics are output to a data file when the program finishes execution. However, to characterize the time varying behavior of the application, these statistics can also be logged periodically to the output file. The interval between printing statistics to file is defined by the user, hence allowing a user to visualize the time varying behavior of an application over its full run. This is useful in identifying representative regions of execution for detailed simulation. In general, statistics are not reset at output time (unless specified), instead the statistics are logged to file and are processed later by a script to determine the application behavior between different logs to the statistics file. We describe in detail the important statistics that can be measured using CMP$im.

- **Application Instruction Profile:** Program instructions are classified into memory and non-memory instructions that belong to different categories of the x86 ISA. Examples of some categories are: branch, jump, ALU, floating-point, semaphore, push, pop, and SSE type instructions.

On a per-cache basis, CMP$im measures a variety of cache statistics, the most important ones being:

- **Cache Accesses/Misses/Writebacks:** Cache accesses are categorized into hits and misses based on the cache access type. We present cache statistics in terms of accesses/1000 instructions, misses/1000 instructions and miss-rate. For writeback caches, the model also tracks the total number of outstanding writebacks and writebacks to subsequent levels of memory.

- **Data Reuse:** A useful metric besides hit/miss information is a measure on the number of times a block is reused after it is allocated in the cache. CMP$im tracks the number of times a block is referenced (on a demand basis) after allocation in the cache. A demand reference is essentially a request initiated by the processor. For example, a load or store access to the cache is a demand request, where as a writeback and remote snoop access is not a demand request. When a block is evicted from the cache, a histogram is maintained using the total number of references to the block before it is evicted. This metric indicates the average number of times a block is reused during its lifetime in the cache.

CMP$im uses the following metrics to measure the degree of data sharing within the application:

- **Shared Cache Line:** For parallel applications, it is useful to measure the amount of data shared between threads. We define a shared cache line as a cache line that is accessed by more than one core during its lifetime in the cache.

- **Shared Access:** An access to a cache line in shared state is defined as a shared access. This metric is a measure of the variation and frequency of accesses to cache lines in shared or private state. It indicates the amount of data sharing in an application which is useful in deciding whether the application would benefit from a shared cache or a private cache.

- **Active-Shared Access:** An active shared access is an access to a shared cache line with the condition that the last core that accessed the shared cache line is different from the current core. For example, if the accesses to a shared cache line are represented by the following core ids: …1, 2, 2, 2, 1, 3, 4, 3, 2, 2, 2, 3, 2…, the accesses by the underlined core IDs are active shared accesses. This metric identifies whether a workload shares cache lines interactively or in a serial fashion. This is useful in deciding whether or not to move cache lines to banks closer to the accessing cores.

CMP$im uses the following metrics to measure coherence traffic on the interconnection network:

- **Coherence Invalidate Frequency:** The coherence invalidate frequency is measured as the total number of invalidate hits to remote cache lines. This metric is measured on a per 1000 instructions basis and is useful for measuring the amount of read-write sharing in an application.

- **Coherence Writeback Frequency:** The coherence writeback frequency is measured as the total number of hits to dirty data in remote private caches. This metric is measured on a per 1000 instructions basis and is useful for measuring the amount of read-write sharing in an application.

Characterization Of Workloads Using CMP$im

Experimental Methodology

Figure 2 illustrates the use of CMP$im to model a four-core CMP 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 assume a zero latency crossbar interconnection network between L2 and L3.

We used CMP$im to characterize the run time memory behavior of the *ammp* workload from the SPECOMP suite with the reference input set [5]. We only capture data cache traffic to measure data sharing between different threads of the workload. The L1 data cache is 32KB, 2-way set associative, with 64B line size and write-through policy. The L2 cache is 256KB, 4-way set associative, with 64B line size and write-back policy. Finally, the L3 cache is 2MB, 8-way set associative, with 64B line size and write-back policy. All caches allocate on a store miss and use the LRU replacement policy. Additionally, inclusion is enforced between all levels of caches and an invalidate-based MESI cache coherence protocol is modeled.

The SPECOMP workload ammp is run with 4 threads on a dual processor system of Intel® Pentium® 4 3.2 GHz processors with hyper-threading enabled. The workload is compiled using the icc compiler, with optimization flags -O3. We ran the workload to completion and logged statistics to file every 10 million instructions executed (on a per thread basis). After simulation, the behavior of the workload over the different intervals of execution is depicted graphically using a post-processing perl script.

Results

The total simulation time to run ammp to completion was approximately 28 hours. This implies that CMP$im can characterize workloads at the speed of ~5 million instructions per second (MIPS). The speed of CMP$im is a function of the memory instruction profile of the workload—the larger the proportion of memory instructions in the workload, the slower the speed of simulation. For a variety of workloads (not described in this paper) we have observed simulation speeds of 4-10 MIPS.

Instruction Profile

Table 1 shows the dynamic instruction profile for the ammp workload when run to completion. The instruction profile reveals that the workload is floating-point intensive with 67% of total instructions composed of x87/floating-point ALU and ALU memory read operations. Furthermore, the workload is memory read-intensive with 30% memory read instructions and 13% memory write instructions. Additionally, the workload has a large basic block length of 10 instructions. This information confirms with existing knowledge of the workload being a floating-point intensive scientific workload.

Cache Performance

Figure 3a 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. To capture the time varying behavior of the workload, we present the phase behavior of the workload at a ten million instruction granularity (represented by the solid lines). The cumulative behavior of the workload is represented by the dashed line. The figure shows that the workload has a periodic pattern with approximately 90 loops. Since the workload exhibits a periodic behavior, we show the cache behavior only for the region between 180 and 250 billion instructions. This region was chosen arbitrarily and the behavior of each loop in this region is representative of all the loops.

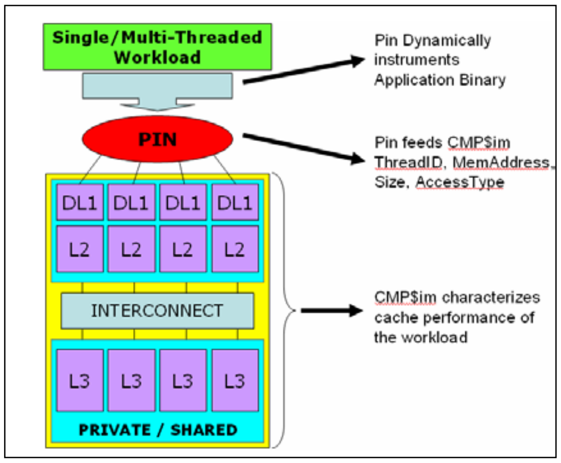
Single/Multi-Threaded Workload
Pin Dynamically instruments Application Binary
PIN
Pin feeds CMP$im ThreadID, MemAddress, Size, AccessType
DL1
DL1
DL1
DL1
L2
L2
L2
L2
INTERCONNECT
CMP$im characterizes cache performance of the workload
L3
L3
L3
L3
PRIVATE / SHARED

Figure 2: Modeling Four Core CMP Using CMP$im

Figures 3b, 3c, 3d, and 3e 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 3d and 3e shows the L3 cache behavior of ammp for

Table 1: Ammp Instruction Profile – Full Run

		THREAD 1		THREAD 2		THREAD 3		THREAD 4		APP TOTAL	
Total Instructions		132,123,667,995		113,030,537,011		114,999,338,301		114,945,398,559		475,098,941,866	
Total Memory Instruction		58,018,005,814		51,334,185,105		52,050,790,913		51,950,310,883		213,353,292,715	
Basic Block Size		9.37		9.57		9.63		9.63		9.54	
X87_ALU	(REG)	45,008,891,419	34.07%	38,513,250,551	34.07%	39,485,874,279	34.34%	39,515,671,299	34.38%	162,523,687,548	34.21%
X87_ALU	(MEM READ)	21,187,241,619	16.04%	18,146,227,341	16.05%	18,607,898,866	16.18%	18,610,410,581	16.19%	76,551,778,407	16.11%
BINARY	(REG)	11,503,086,485	8.71%	9,697,294,457	8.58%	9,802,889,596	8.52%	9,832,198,356	8.55%	40,835,468,894	8.60%
DATAXFER	(MEM READ)	9,605,654,652	7.27%	8,515,068,597	7.53%	8,515,962,988	7.41%	8,450,249,019	7.35%	35,086,935,256	7.39%
X87_ALU	(MEM WRITE)	9,164,363,756	6.94%	7,989,095,413	7.07%	8,145,505,894	7.08%	8,154,258,765	7.09%	33,453,223,828	7.04%
COND_BR	(REG)	8,858,499,892	6.70%	6,699,980,847	5.93%	6,839,450,084	5.95%	6,847,925,987	5.96%	29,245,856,810	6.16%
PUSH	(MEM WRITE)	4,523,982,284	3.42%	4,265,431,824	3.77%	4,242,344,381	3.69%	4,233,075,058	3.68%	17,264,833,547	3.63%
DATAXFER	(REG)	4,454,015,006	3.37%	4,004,293,102	3.54%	3,998,893,624	3.48%	3,986,456,424	3.47%	16,443,658,156	3.46%
BINARY	(MEM READ)	3,121,322,351	2.36%	2,614,733,809	2.31%	2,747,625,151	2.39%	2,743,938,827	2.39%	11,227,620,138	2.36%
POP	(MEM READ)	2,310,376,781	1.75%	2,236,817,478	1.98%	2,236,050,749	1.94%	2,225,913,848	1.94%	9,009,158,856	1.90%
RET	(MEM READ)	2,271,263,397	1.72%	2,235,578,040	1.98%	2,235,219,695	1.94%	2,224,997,245	1.94%	8,967,058,377	1.89%
CALL	(MEM WRITE)	2,271,201,992	1.72%	2,235,579,957	1.98%	2,235,163,991	1.94%	2,224,941,665	1.94%	8,966,887,605	1.89%
PUSH	(MEM R/W)	1,613,424,472	1.22%	1,648,479,385	1.46%	1,639,221,933	1.43%	1,635,809,704	1.42%	6,536,935,494	1.38%
DATAXFER	(MEM WRITE)	1,583,962,677	1.20%	1,216,772,167	1.08%	1,217,571,905	1.06%	1,218,836,798	1.06%	5,237,143,547	1.10%
LOGICAL	(REG)	1,495,860,679	1.13%	810,437,292	0.72%	824,143,508	0.72%	820,805,504	0.71%	3,951,246,983	0.83%
MISC	(REG)	1,486,349,141	1.12%	913,733,382	0.81%	940,356,227	0.82%	937,079,994	0.82%	4,277,518,744	0.90%
UNCOND_BR	(REG)	703,977,798	0.53%	639,406,862	0.57%	637,836,123	0.55%	637,898,672	0.55%	2,619,119,455	0.55%
FLAGOP	(REG)	321,324,917	0.24%	146,462,630	0.13%	145,426,503	0.13%	145,758,671	0.13%	758,972,721	0.16%
BITBYTE	(REG)	246,391,743	0.19%	271,208,965	0.24%	273,535,070	0.24%	271,150,426	0.24%	1,062,286,204	0.22%
BINARY	(MEM R/W)	245,668,912	0.19%	229,995,484	0.20%	228,048,949	0.20%	227,703,835	0.20%	931,417,180	0.20%
MISC	(MEM READ)	73,631,329	0.06%	282,829	0.00%	110,888	0.00%	110,357	0.00%	74,135,403	0.02%
LOGICAL	(MEM READ)	43,336,919	0.03%	31,255	0.00%	2,618	0.00%	2,400	0.00%	43,373,192	0.01%
SHIFT	(REG)	27,265,004	0.02%	282,518	0.00%	141,002	0.00%	140,986	0.00%	27,829,510	0.01%
SHIFT	(MEM R/W)	1,224,544	0.00%		0.00%		0.00%		0.00%	1,224,544	0.00%
BITBYTE	(MEM READ)	843,786	0.00%		0.00%		0.00%		0.00%	843,786	0.00%
LOGICAL	(MEM R/W)	437,497	0.00%	1,302	0.00%	1,373	0.00%	1,358	0.00%	441,530	0.00%
CALL	(MEM R/W)	66,318	0.00%	59,078	0.00%	59,133	0.00%	59,024	0.00%	243,553	0.00%
UNCOND_BR	(MEM READ)	2,528	0.00%	31,146	0.00%	2,399	0.00%	2,399	0.00%	38,472	0.00%
ROTATE	(REG)	97	0.00%	1,300	0.00%	1,372	0.00%	1,357	0.00%	4,126	0.00%
Total Memory Reads		38,613,673,362	29.23%	33,748,770,495	29.86%	34,342,873,354	29.86%	34,255,624,676	29.80%	140,960,941,887	29.67%
Total Memory Writes		17,543,510,709	13.28%	15,706,879,361	13.90%	15,840,586,171	13.77%	15,831,112,286	13.77%	64,922,088,527	13.66%
Total Memory Read/Writes		1,860,821,743	1.41%	1,878,535,249	1.66%	1,867,331,388	1.62%	1,863,573,921	1.62%	7,470,262,301	1.57%
Total Control Instructions		14,104,945,607	10.68%	11,810,576,852	10.45%	11,947,672,292	10.39%	11,935,765,968	10.38%	49,798,960,719	10.48%

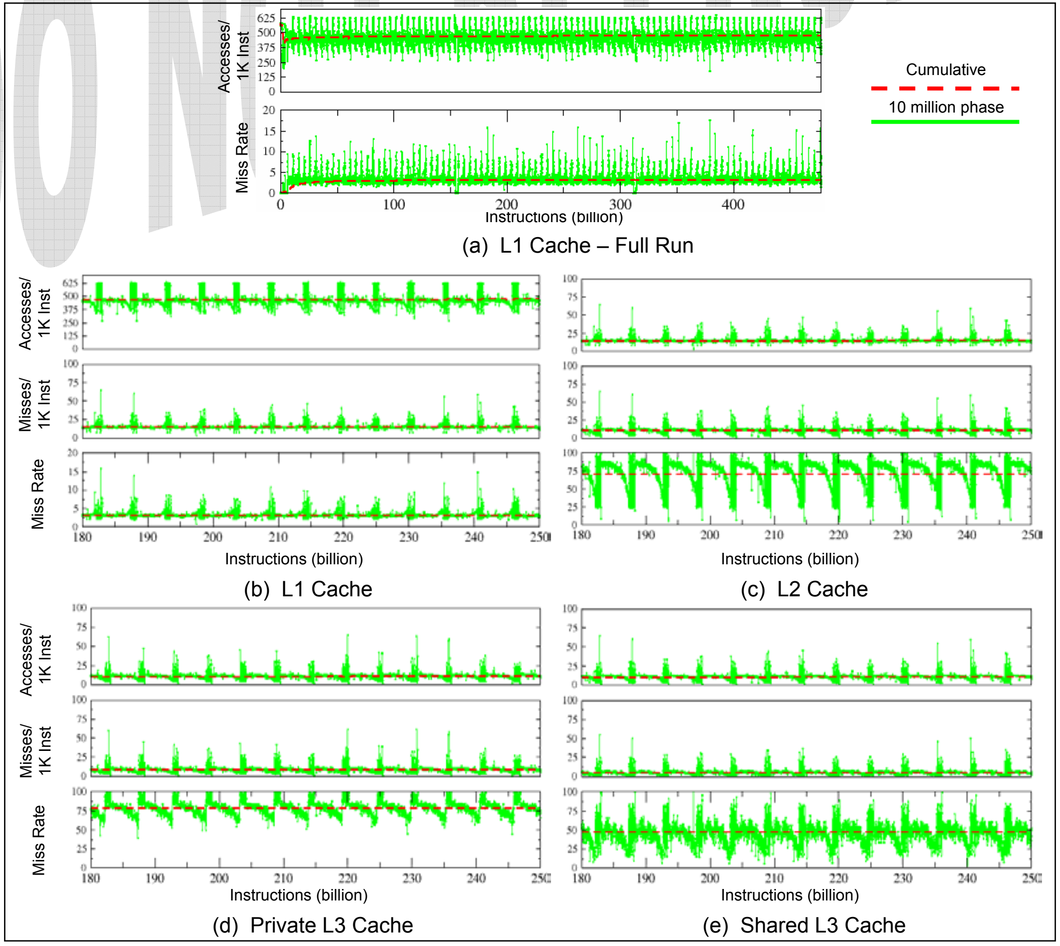

Figure 2: Ammp Cache Performance

private and shared 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 avoids duplication by having a centralized copy of the line, which increases the effective cache capacity and reduces miss rate. In a later section, we show that ammp benefits from a shared cache primarily because of extensive data sharing.

CACHE REUSE

Figures 4a, 4b, 4c, and 4d show the reuse distribution of lines evicted from the L1, L2, and L3 cache. Recall that reuse is measured as the total number of times a cache line is $_g$referenced by a demand access *after* it is installed. Let reuse be defined by the symbol: ρ. Reuse values are classified as ρ =0, 1, 2, 3, 4, 5, 6, and 7, and regions $8 \le \rho < 16$, $16 \le \rho < 32$, … $4096 \le \rho < 8192$, and $\rho \ge 8192$. The figures show reuse value of zero at the bottom of the graph and a reuse value of 8192 or higher at the top of the graph (We recommend viewing a soft copy or a color print out of this graph). On average, the figure shows that 76%, 12%, 10%, and 30% of lines evicted from the L1, L2, private L3, and shared L3 respectively are reused at least once. The low value of reuse of lines in the L2 and L3 cache is due to a high miss rate causing lines to

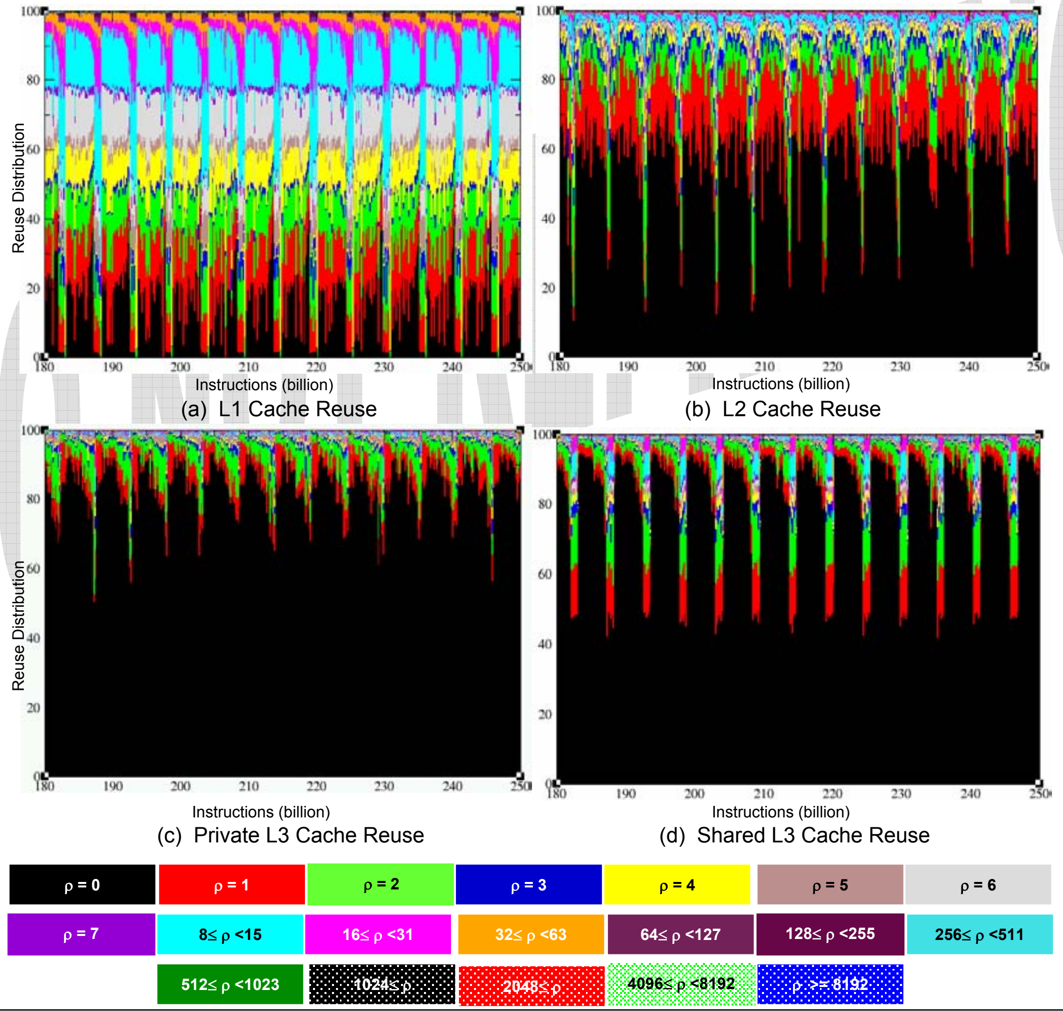

Figure 4: Ammp Reuse (ρ) Distribution

be evicted before they get referenced again. In addition, the poor reuse of data in the shared L3 cache is because temporal locality of data is short lived. The low value of reuse indicates an inefficient use of the cache. Since 5–95% of the lines evicted from the different levels of cache hierarchy are never referenced after installation, implying that cache performance can be improved by not installing these lines. This motivates investigating alternative cache allocation policies to reduce cache pollution and improve effective cache capacity.

DATA SHARING

Figure 5a 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 5a shows that half the cache is shared by two or more cores. Figure 5b provides more insight into the frequency of access to shared data by presenting the distribution of accesses to the shared last-level cache. The figure shows as much as 50-80% of the cache accesses are to lines that are shared by two or more cores. Furthermore, Figure 5c also shows that 50-80% of accesses are to cache lines that are actively shared. Thus, ammp exhibits a significant amount of data sharing between multiple cores because of which it significantly benefits from a shared cache compared to a private cache.

Coherence Traffic

Figure 5d presents the coherence invalidate frequency and the coherence writeback frequency for the L1 and L2 cache. The figure shows that cache lines are invalidated at a rate of 0.1 to 1.5 and 0.1 to 3 invalidates per 1000 instructions from the L1 and L2 cache respectively. The number of invalidates are higher in the L2 cache because the size of the L2 is larger than the L1.

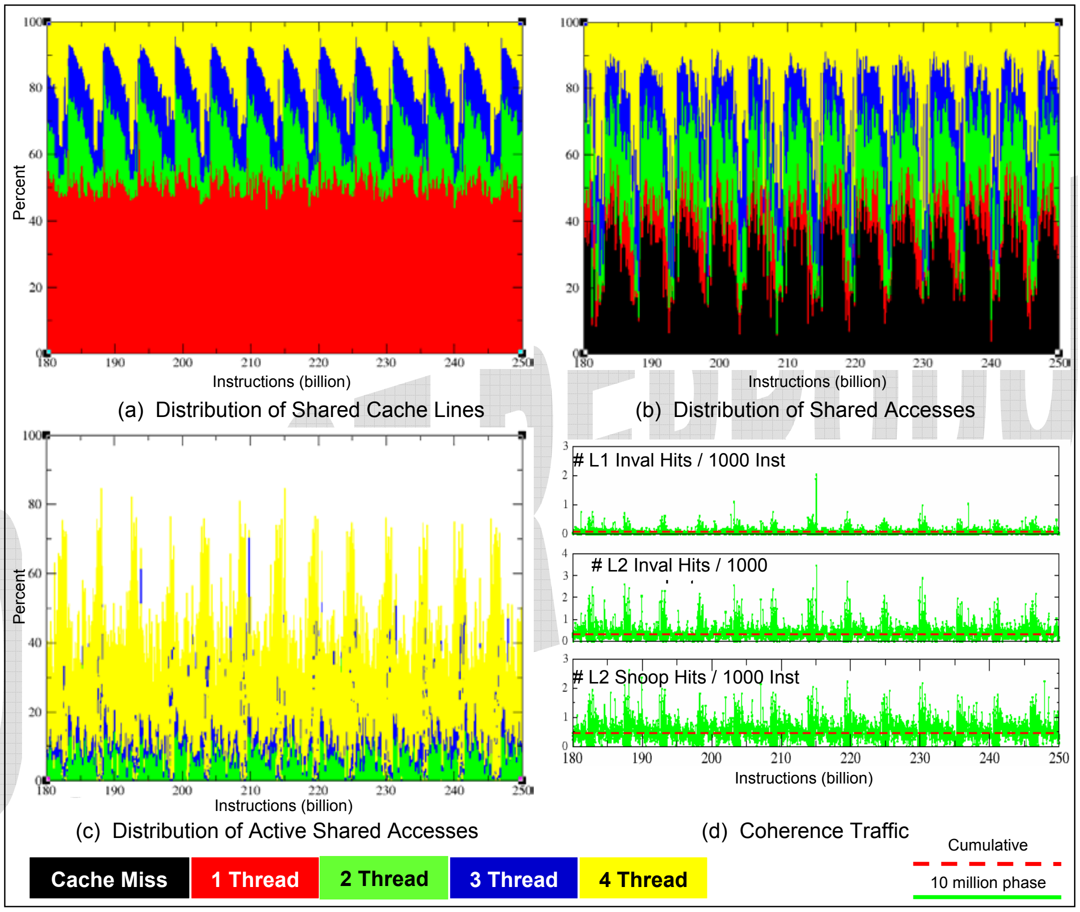

Figure 5: Ammp Sharing Statistics

In a given interval, a large number of snoop hits and invalidate hits indicates a large amount of read/write data sharing. The figure shows that the beginning of each loop exhibits read/write sharing due to the existence of snoop hits (0.1 to 2 hits per 1000 instructions) and invalidate hits (0.1 to 3 invalidates per 1000 instructions). The interconnection network is stressed heavily at the beginning of each loop due to the large amount of read/write sharing. On the other hand, the interconnection network is not stressed during the rest of the loop due to very little read/write sharing. Thus, even though the average bandwidth demand is low, a high bandwidth interconnection network is required to handle the clustered coherence traffic.

Conclusions

This paper proposes the use of binary instrumentation to conduct memory performance studies as an alternative to execution-driven and trace-driven methodologies. Using the binary instrumentation tool 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 measure the dynamic instruction profile and memory system performance of a multi-threaded workload, *ammp*, when run to completion. Our experience shows that binary instrumentation based simulation allows for full characterization of workloads run to completion at speeds ranging from 4-10 MIPS.

On-Going Work

Speculation

The work presented in this paper assumes no speculative execution, i.e. perfect branch prediction. However, with binary instrumentation based simulation (using Pin) it is also possible to model wrong-path execution. This is accomplished by modeling a branch predictor and executing instructions down a path predicted by the branch predictor. To correctly model the effects of wrong-path execution using binary instrumentation, the binary instrumentation tool must allow a user to create check points and change architectural state. This is required if the branch predictor makes an incorrect decision resulting in wrong-path execution. To resolve wrong-path execution, after some delay (as specified by the user), the checkpoint must be restored to allow correct program execution. The addition of speculative execution in our CMP$im infrastructure is part of our on-going work. We would like to mention here though that the miss rates on a Pentium® III (using the PAPI performance counter suite) and a simulation of the P3 memory hierarchy (using CMP$im) revealed that the cache miss rates with CMP$im were within 5-10% of performance counter values for various single-threaded workloads.

TIMING MODELS

The use of binary instrumentation can also be used to perform processor performance/sensitivity studies. For example, one can model an in-order processor using the CMP$im simulator. By assigning a latency of execution T_c for an instruction belonging to instruction category c, penalty T_b for a branch misprediction, access latency T_l for level l in the cache hierarchy, and access latency of T_m to memory, one can estimate the performance of a workload using the following equation:

$$\alpha_c T_c + \beta T_b + \Sigma A_l T_l + \eta T_m$$

Where α is the number of instructions executed, β is the number of branch mispredictions, A is the number of accesses, and η is the number of misses to memory. Note that this simple in-order timing model assumes stall on miss and not stall on use.

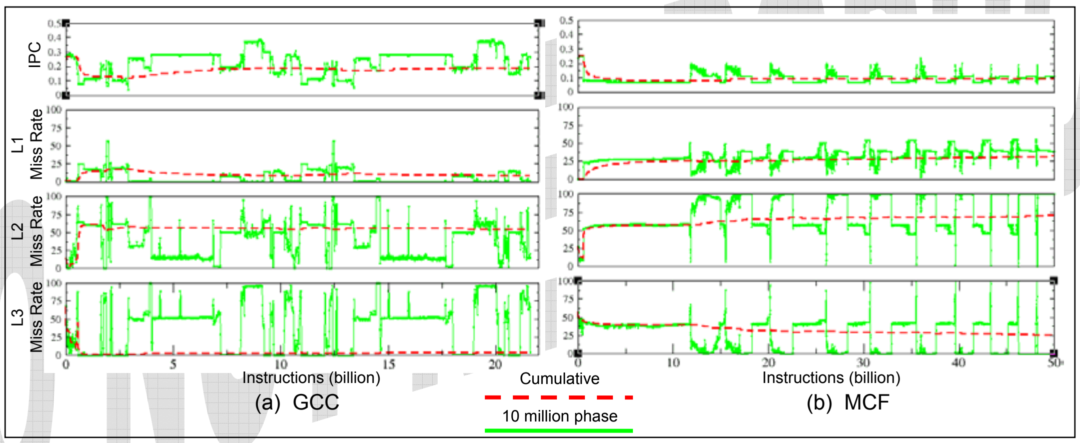

Figure 5: Performance Characterization of Workloads Using CMP$im

Using the above simple timing model, and assuming single cycle execution for all instructions, perfect branch prediction (β = 0) and latencies of 3, 10, and 35 to the 1st level, 2nd level, and 3rd level cache, and finally 100 cycles of latency to memory, we present in Figure 6 the instructions per cycle and the L1, L2, and L3 cache miss rates of gcc and mcf from the SPEC2000 benchmark suite.

From Figure 6a, we observe that gcc has several phases of execution which exercise the different levels of the cache hierarchy. As expected, the figure shows that gcc performs best when it fits well in the L1 cache. When it does not fit in the L1 cache, we observe phases of execution (~1.1 and ~11.5 billion instructions) where the performance of gcc is dependant on fitting in the L2 cache. In general, based on the figure, we conclude that the performance of gcc is sensitive to L2 cache size. Similarly, from Figure 6b, we observe that mcf exhibits loop behavior in the last 40 billion instructions executed. The figure shows phases of execution where the working set of mcf is larger than L2 hence we see periods of execution where the miss rate of L2 is a 100%. In general, we observe separate phases of execution where the performance of mcf is sometimes dependent on the miss rate of the L2 cache and sometimes on the miss rate of the L3 cache.

The above rudimentary timing model can be extended to incorporate dependency between instructions and analytically model stall-on-use in-order processors. Furthermore, the CMP$im infrastructure can also be used to analytically model an out-of-order processor as proposed by Karkhanis et al [25]. These analytical models work well on first-order with single-threaded workloads or one thread of a multi-threaded workload. However, estimating the performance of a multi-threaded workload is non-trivial as there is no notion of time (to measure inter-thread contention) with binary instrumentation based simulation. This is part of our on-going work.

OTHER MEMORY HIERARCHY STUDIES

Besides cache performance studies, CMP$im can also be used to explore the design space of TLBs and data prefetching. We are also using CMP$im to investigate novel cache replacement policies to improve the reuse of lines in the cache. Finally, we are also extending CMP$im to study quality of service studies when executing multi-programmed workloads on CMPs.

BIBLIOGRAPHY/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] 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.

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

[8] R. Alameldeen and D. A. Wood. "Variability in Architectural Simulations of Multi-threaded Workloads." In Proceedings of the International Conference on High Performance Computer Architecture (HPCA), Anaheim, California, 2003.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] E. Speight, H. Shafi, L. Zhang, R. Rajamony. "Adaptive Mechanisms and Policies for Managing Cache Hierarchies in CMPs." In Proceedings of the 32nd International Symposium on Computer Architecture (ISCA), Wisconsin, Madison, 2005.

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

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

[24] 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.

[25] 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.