Design Exploration of an Instruction-Based Shared Markov Table on CMPs

Lixin Su & Karthik Ramachandran

Department of Electrical and Computer Engineering
University of Wisconsin
Madison, WI 53706

lsu@ece.wisc.edu ramachan@cae.wisc.edu

Abstract

Our project starts from investigating if instruction-based sharing exists on commercial workloads such as Apache, Zeus, Jbb, and Oltp running on CMPs. We find that there is a large amount of instruction-based sharing on CMPs. Constructive interference at the instruction cache miss level also exists among different CMP cores. We further study if the implementation of a shared Markov table can help reduce L1 instruction cache misses for each CMP core. We find that a reasonably small shared Markov table, varying from 4K entries to 32K entries, can help reduce L1 instruction cache misses and can potentially evenly increase each CMP core's performance and thus the overall CMP performance.

KEYWORDS: CMPs, commercial workloads, Markov Table, Instruction Sharing

Design Exploration of an Instruction-Based Shared Markov Table on CMPs

Abstract

Our project starts from investigating if instruction-based sharing exists on commercial workloads such as Apache, Zeus, Jbb, and Oltp running on CMPs. We find that there is a large amount of instruction-based sharing on CMPs. Constructive interference at the instruction cache miss level also exists among different CMP cores. We further study if the implementation of a shared Markov table can help reduce L1 instruction cache misses for each CMP core. We find that a reasonably small shared Markov table, varying from 4K entries to 32K entries, can help reduce L1 instruction cache misses and can potentially evenly increase each CMP core's performance and thus the overall CMP performance.

1 .0   Introduction

Modern integrated circuit and packaging technologies have led to several competing high-performance microprocessor architectures, superscalar processors (OoOs) [8,12], very-long-instruction-word processors (VLIWs) [9,10,11], simultaneous multithreading processors (SMTs) [1,2] and chip multiprocessors (CMPs) [4,5,6]. Superscalar processors typically use wide fetching, out-of-order issuing and execution, and multiple execution units to achieve high performance. VLIWs can efficiently utilize dynamic-static interface and take advantage of compiler technologies to increase computing throughput. SMTs dissect applications into hardware threads and alternate the execution of these threads when some threads experience long stalls. As a newly emerging microprocessor architecture, CMPs implement multiple processing cores on the same chip and they act like an onchip mini-multiprocessor. CMPs employ the typical divide-and-conquer engineering approach. Each processing core can be independently designed and then replicated during the final chip integration phase. This approach greatly reduces the design complexity and shortens the overall chip design cycle. In addition, CMPs can reuse in a new on-chip environment many existing multiprocessor research fruits, esp. the existing multiprocessor protocols. The on-chip multiprocessing brings higher communication bandwidth, shorter access latencies, and better wiring options, leading to better interprocessor interactions than existing multiprocessors that are limited by offchip technology constraints. Furthermore, CMPs can still be compatible with the other three competing technologies: superscalars, VLIWs, and SMTs. Each CMP core can be implemented as superscalars, VLIWs, or SMTs. Other than implementing complex processing cores, CMPs can choose a simple in-order processing core design and save precious on-chip resources for a better memory system design.

Meanwhile, commercial workloads [15] such as databases and Web applications have surpassed scientific computing workloads to become the largest and fastest-growing market segment for high performance severs. Such workloads contain multiple processes, provide accesses to multiple users, and run on multiple processors. Database and Web applications can run similar transactions concurrently, process identical queries

from different users, and schedule executing threads in case of lock contentions. Therefore, commercial workloads should have some inherent interactions among concurrent threads and transactions and these interactions can behave externally on the interactions among different CMP cores' processing. In addition, most commercial workloads still stay in a coarse grained level that requires a large computation to communication ratio in order to better overlap computations with communications. However active research is being conducted to integrate fine grained parallelism into existing commercial applications [13]. This trend will lead to even closer interactions among internal threads in commercial workloads and thus the closer interactions between workloads and CMPs.

One fundamental approach towards better understanding the interactions between commercial workloads and CMPs is studying if sharing exists among commercial workloads' computing tasks running on different CMP cores. Such sharing might exhibit as a thread-level sharing, an instruction-level sharing, or a data-level sharing. The thread-level sharing includes both the instruction-level sharing and the data-level sharing but it shows in a broader scope. The data-level sharing might often accompany with the instruction-level sharing and can be the outcome of the instruction-level sharing. As a starting point, our projects is focused on the instruction-level sharing study. We will identify if such a sharing exists for commercial workloads running on CMPs. We will continue to study if the instruction-level sharing can be utilized to improve overall CMP performance if it does exist. Sharings can only help improve CMP performance if sharings can be converted to constructive interferences for tasks running on different cores.

Our study started from counting repeated times of sequences of instructions. We found that short sequences of instructions repeat frequently across CMP cores. This indicates the existence of instruction-level sharing. After that, we moved on to instruction cache miss sharing study. We implemented a shared Markov table to store L1 instruction cache misses from 16 CMP cores. We studied the hit rate and size of such a shared table. We found that instruction-level sharing exists in L1 instruction cache miss streams from different CMP cores and a small shared Markov table can help dramatically reduce the number of L1 instruction cache misses and thus improve CMP performance.

The rest of this paper is organized as follows. Section 2 presents an overview of a CMP. Section 3 provides an insight to the current class of applications and workloads. Section 4 describes an overview of the project and the simulation environment. Design and simulation results of our first experiment are presented in Section 5. In Section 6 we describe our Design and simulation results of the second experiment. Finally, we discuss related work and conclude.

2 .0   CMPs - An Overview

The main motivation for building a single chip multiprocessor comes from two sources: there are both a technology push and an application pull. CMOS technologies have given us a large amount of on-chip resources

and in the near future there may be one billion transistors on a single chip. How to efficiently utilize these transistors will become an important issue that architects need to address. Aggressive superscalars with wide instruction issuing and large amounts of speculation support are leading us to a dead-end: the increase of the delay in the complex issue queue and the necessity of supporting multi-ported register files. The complexity of the bypass logic also grows quadratically with the number of execution units. VLIWs are being used in Intel's Itanium processors but they require large amounts of support from software, e.g., both operating systems and compilers, which may require rewriting and recompilation of existing applications. This approach is being proven to be unpopular by the Itanium family processors. SMTs are putting a lot of pressure on register files and cause challenge to chip designs. The existing SMTs can only support a maximum of two threads and the two threads can only commit instructions that don't affect the other thread, which reduces the ROB design complexity but lower the performance benefit of SMTs. CMPs, as a newly emerging competing technology, put multiple processing cores on a single chip to reduce design complexities and explore multithreaded parallelism in applications. CMP cores can be either homogeneous or heterogeneous. CMP cores can also be implemented as simple in-order cores or less aggressive OoO cores. It might also be possible for CMP cores to be designed as SMT cores or VLIW cores if the combinations can be proven to dramatically increase overall system performance.

The primary benefits from integrating modules onto a single chip arise from both the reduction of design complexities and more efficient communication interfaces. First, CMP cores are much simpler than aggressive superscalar processors. They can be in-order and won't hurt performance due to the existence of multiple cores. CMP cores can be designed once and then replicated if homogeneous cores are used. Second, there are fewer signals that cross chip boundaries, leading to lower latency communications than existing multiprocessors. Third, integrations allow for substantially higher communication bandwidth by removing constraints imposed by scarce external pin resources. Intrachip wiring among processing cores can also lead to better bandwidth for interprocessor communications. Both lower communication latencies and larger communication bandwidth present us opportunities for sharing different processing cores' knowledge.

CMPs can also integrate a large portion of memory hierarchy to the chip, e.g., L1 caches and L2 caches. Integrations of multiple processing cores and memory hierarchy can help achieve high performance. Other than that, the interface between CMP cores and memory hierarchy stay very similar to the interface on existing multiprocessor systems except that communication latencies are much shorter and communication bandwidth is significantly larger. This leads to both the opportunities of reusing multiprocessing coherence protocols and the challenge of improving existing protocols in the presence of better communication interfaces.

3 .0   Commercial Workloads

From the application perspective, microarchitectures have to be designed to suit the inherent characteris-

tics of applications to maximize microarchitectures' performance. For CMPs designed to run on servers, most applications will be commercial workloads. These workloads are different from scientific computing workloads. First, commercial workloads are multithreaded, multitiered, and multiuser-oriented. Existing applications are designed with a coarse grained parallelism. The computation to communication ratio is generally high in order to overcome the synchronization cost. Future applications will shift toward a finer grained parallelism. A fine grained parallelism can better utilize intrachip communication interfaces presented by CMPs. Second, many commercial workloads are database and Web applications. Concurrent transactions and queries can be launched by users. These transactions and queries can often use similar code and process similar data tables in databases. This leads to the possibility of instruction sharing among different processor cores. Third, most computing in commercial workloads are integer operations. Floating-point application performance is less important for commercial workloads.

4 .0   Project Overview

Our project can be divided into two phases. First, we need to prove if instruction sharing exists on commercial applications running on CMPs. In this phase, we want to prove the existence of instruction sharing in two levels -- instruction streams executed by different CMP cores and instruction cache miss streams experienced by different cores. The instruction stream study is supposed to be simple and fast, which gives us a glimpse at the possibility of the existence of the instruction sharing. The instruction cache miss study is conducted in detail and is combined with the design exploration of shared Markov table under L1 instruction caches. The second phase is a thorough design exploration of our proposed shared Markov table. Here we mainly focus on the prospective that a shared Markov table can reduce the number of instruction cache misses for each CMP core and how the table should be configured, e.g., the size and teh associativity of the table. We also want to investigate the interaction of a shared Markov table with other on-chip hardware structures, e.g., shared L2 caches and sequential prefetchers, that can help reduce the number of instruction cache misses.

A shared Markov table under L1 instruction caches can help reduce instruction cache misses and improve performance only if it meets the following requirements. First, the hit rate of lookups by each CMP core has to be high. This requirement equals to that constructive interference from multiple CMP cores has to exist. Second, the size of the Markov table should be reasonably small so that access latencies are lower compared with access latencies to L2 caches.

We use the Simics full system simulator for our study. Simics with Multifacet extensions is an execution driven simulator and it simulates a SPARC based CMP. We focus on four commercial workloads: SPECJbb, Apache, Zeus and Oltp. SPECJbb is a java version integrated database application.

Apache is static application server workload. Zeus is a dynamic application workload. Oltp is tpc-c like database transaction application.

5 .0   Experiment I

This is our first experiment and it largely decides whether we can proceed in the direction we planned. Therefore, our first experiment needs to be simple and give us a general idea whether it is possible for the instruction sharing to exist on CMPs. If it doesn't, our hypothesis is wrong and we probably need to change our project. If it does exist, we will move further to the instruction cache miss sharing study and we will investigate the feasibility of implementing an instruction-based shared structure such as Markov tables for CMPs.

5.1   Experiment Setup

We designed a very simple experiment as our experiment I. With this experiment, we simply count the repeated times of a sequence of n consecutive instruction across different CMP cores. If the repeated times are very large, it might be very promising that instruction sharing exists and we can continue with our planned study.

Processor	P1	P2	P3	P4
Sequence of 2 I's	{A,B}... N times	{A,B}... N times	{A,B}... N times	{A,B}... N times
Count on each P	N-1	N-1	N-1	N-1

TABLE 1. An example of how we count the repeated times of a sequence of n instructions. For this case our counter is 4N, which means that a sequence of 2 instructions repeats 4N times across 4 processors.

Table I is an example showing how we count the repeated times of a sequence of n instructions. Say a particular sequence {A, B} occurs N times on P1 through P4 on a 4 core CMP system, then we say that the individual repeated counts on each processor is N-1 and the total count of the repeated sequence on the entire CMP is 4N.

5.2   Simulator Configuration

For our experiment we use Simics full system simulator with Opal and Ruby extensions. For simplicity we only look at CMPs with 16 on-chip cores. In addition, we only have 16p checkpoints for our workloads: Zeus, Jbb, Apache and Oltp.

5.3   Results

As previously mentioned, the main goals of our first experiment were to investigate into the instruction-sharing pattern on our CMP and give us a possibility if instruction sharing exists on commercial workloads running on CMPs. Figure 1 gives us an insight to the amount of repeated sequences occurring on our system. We observe that the number of times a sequence pattern from two to five instructions repeat on the entire system is nearly 75%. This was found across all the four benchmarks and for each

transaction. As per our design, we clear out statistics after each transaction that consists of 20000 instructions, hence we find the count to have an almost horizontal slope for each workload. Initially this data seemed to be weird to us due to the high repetition count and also due to the not so big difference in the number of times a sequence pattern of 5 instructions occurring from a sequence pattern of 2 instructions. But on further analysis we thought that spin loops could be the cause for such high counts. We added some more logic to count the percentage of spin loops occurring and in commensurate with our hypothesis it was interesting that spin loops occurred 50% of the time for a non warm-up cache and around 30% of the time with a warmed-up cache. The wisdom of pushing for a shared I-Cache was still naïve. Hence we moved on to our design of second experiment that focuses on the I-Cache miss-sharing pattern across our CMP system.

6 .0   Experiment II

Our experiment I results led us further towards the belief of our hypothesis that the instruction sharing exists across different processor cores. However, experiment I itself is yet enough to prove that our hypothesis is a fact and designing a shared structure to increase computing performance on CMPs can be a feasibility. Our experiment II is a natural follow-up experiment from the first one and is a major part in our project. It is designed and set up to prove that the instruction sharing does exist across CMP cores. The experiment focuses on the first level instruction cache miss study and tries to answer if the instruction sharing exists within CMPs. The experiment also tries to answer questions if a shared Markov table can help significantly reduce the number of the first level instruction cache misses for almost all CMP cores and potentially increase CMP's overall performance.

6.1   Experiment Setup

In the experiment, we implemented a 16K-entry fully associative shared Markov table in Opal. See figure 2 for details. All the 16K entries are maintained in an LRU list. The LRU head entry represents an entry that is most recently visited and the LRU tail entry represents an entry that is least recently visited. Each entry has an LRU distance from the LRU head, which is exactly how many entries this entry is away from the LRU head entry. The LRU tail entry is replaced when the table is full and a new entry is to be inserted. Each entry can hold two consecutive instruction cache misses from the same processor core. An atomic lookup in the Markov table is performed when a processor sees two consecutive misses from its own instruction cache. The hit or miss counter is respectively incremented by one depending on whether the lookup turns out to be a hit or miss. A new entry will be inserted to the LRU tail if the lookup is a miss. The new entry holds the misses the lookup processor just experienced. The old LRU tail entry has to be replaced if the table is full. Similarly, the LRU distance of the hit entry is recorded in a histogram class if
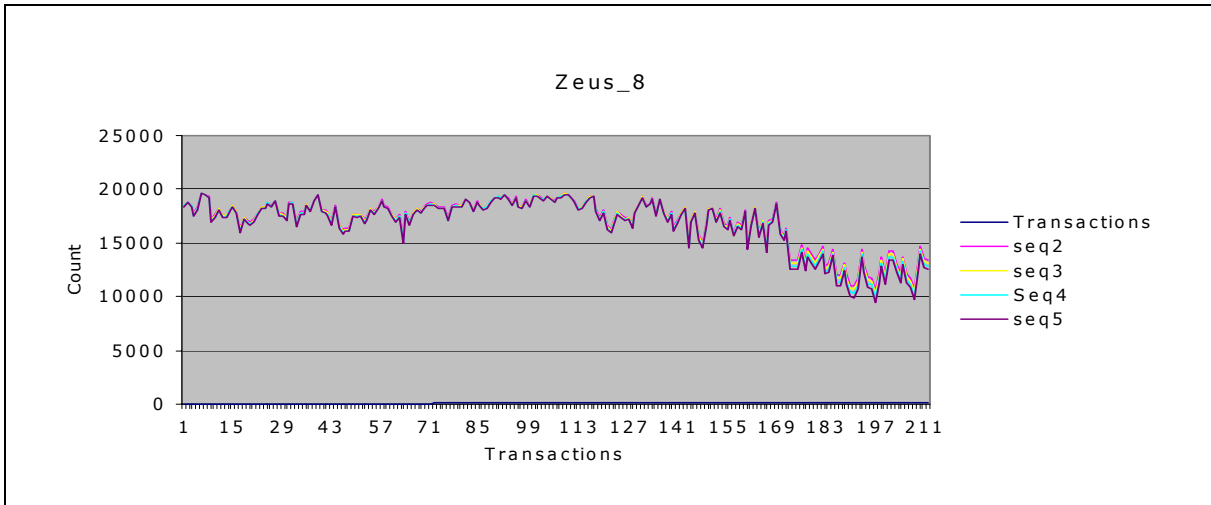
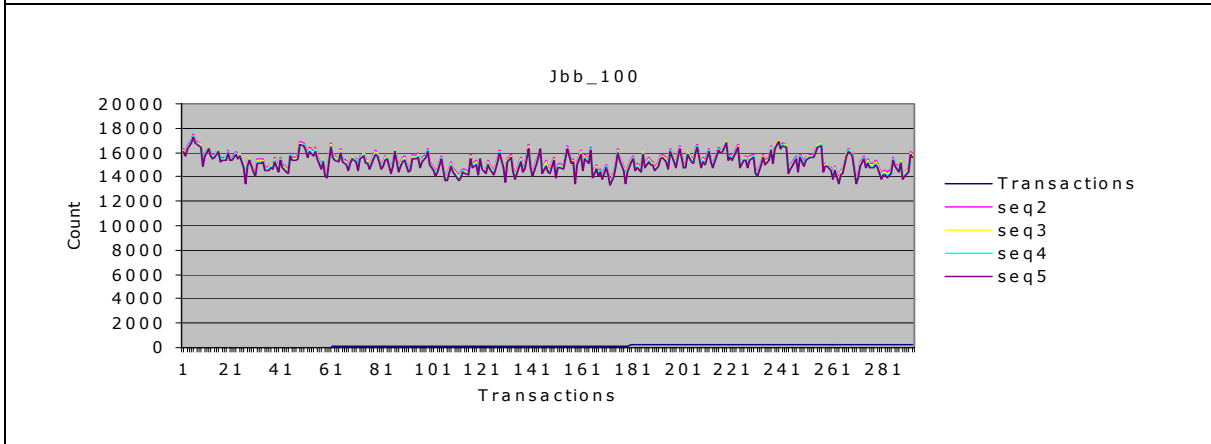Figure 1 (a) Repeated times of sequences of 2, 3, 4, 5 instruction across 16 CMP cores for Zeus.

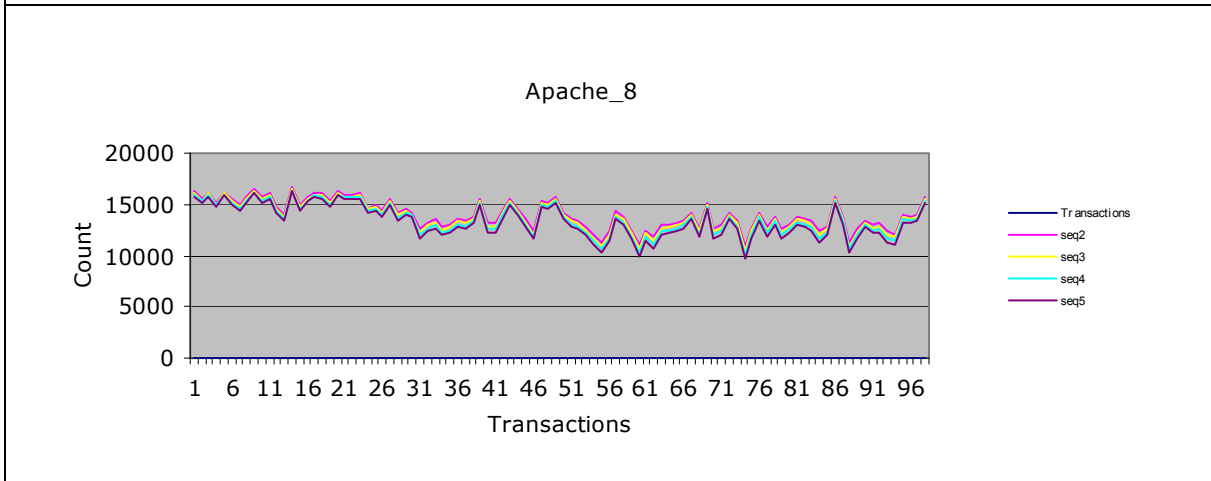Figure 1 (b) Repeated times of sequences of 2, 3, 4, 5 instructions across 16 CMP cores for Jbb.

Apache_8

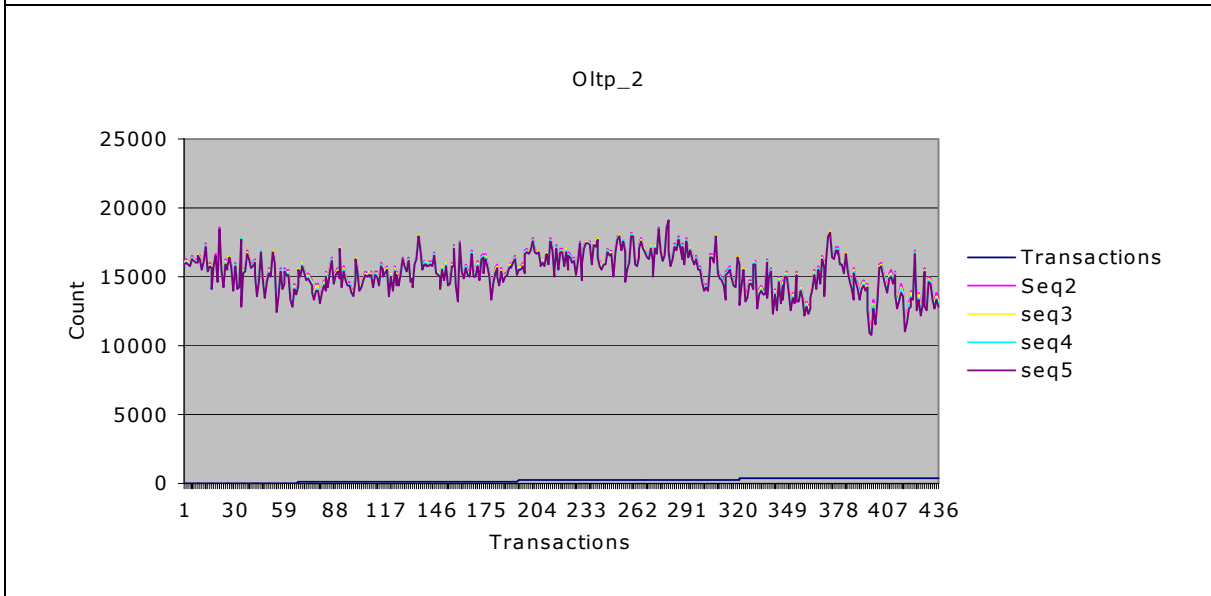Figure 1 (c) Repeated times of sequences of 2, 3, 4, 5 instructions across 16 CMP cores for Apache.

Oltp_2

Figure 1 (d) Repeated times of sequences of 2, 3, 4, 5 instructions across 16 CMP cores for Oltp.

the lookup is a hit. The hit entry is automatically moved to the head of the LRU list

6.2 Simulator Configuration

Our simulation is based on Simics full system simulator from the Multifacet group. It uses both Opal and Ruby. Opal simulates the processor core and Ruby simulates the memory system. Our code is written in Opal. Table 2 listed below shows some important simulation parameters. Our instruction cache configuration is aggressive compared with the current L1 instruction cache configuration for uniprocessors. CMPs integrate multiple processor cores on the same chip and each core has an L1 instruction cache and an L1 data cache. The real estate for each core might be even precious than for uniprocessors. Therefore, we predict that first level instruction cache tend to remain direct mapped 32KB caches in the near future. However, the simulator only allows us to choose one single configuration for both instruction
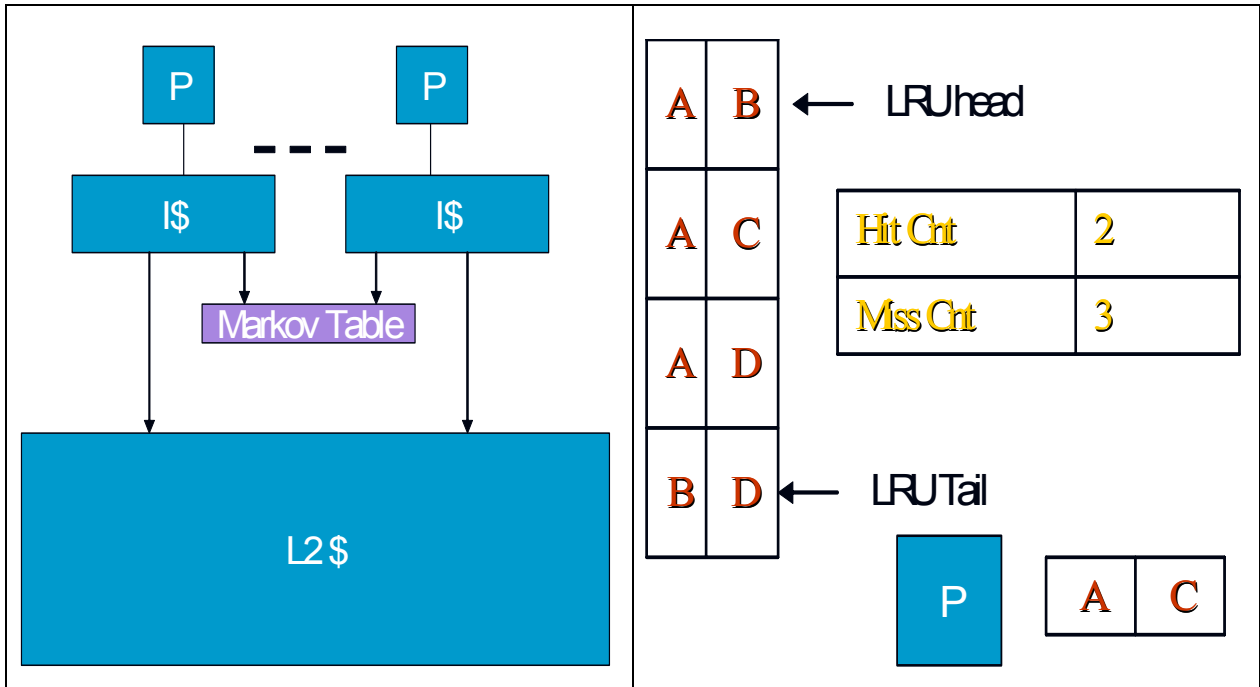
Figure 2 (a) Left figure. The hardware design block diagram. A small shared Markov is shared by all the processor cores within CMPs. (b) Right Figure. The internal structure of a shared Markov table. The red fonts represent cache misses. All the entries within the Markov table are maintained in an LRU list. There are two counters: hit counter and miss counter. A processor performs an atomic lookup when it sees two consecutive instruction cache misses. We only show one processor in the figure.

caches and data caches. We choose 2-way associative 64KB cache configuration to leverage the benefit of a bigger L1 data cache.

CMP cores	16
I cache (size/associativity/block size)	64KB/2/64B
D cache (size/associativity/block size)	64KB/2/64B
Shared L2 cache (size/associativity/block size/banks)	1MB/4/64B/16
Protocol	MSI_dir_L1_MOSI_dir_L2_CMP

TABLE 2. Experiment II simulator parameters.

6.3   Hit Ratio in the Shared Markov Table

We first investigate the hit rate of all the lookups in the shared Markov table in order to find if the instruction sharing exists across CMP cores and if constructive interference exists across different CMP cores' instruction cache miss streams. We define the hit rate as the number of hits in the Markov table divided by the number of lookups performed by all CMP cores. We also run the same experiment for the uniprocessor where only one CMP core is connected to a Markov table and only this processor can look up in and update the Markov table. We were unable to examine the hit rate for a "pure" uniprocessor since we were only given 16p checkpoints. But we think the instruction cache miss stream from one CMP core out of sixteen homogeneous cores should behave similarly to the instruction cache miss stream of a uniproces-
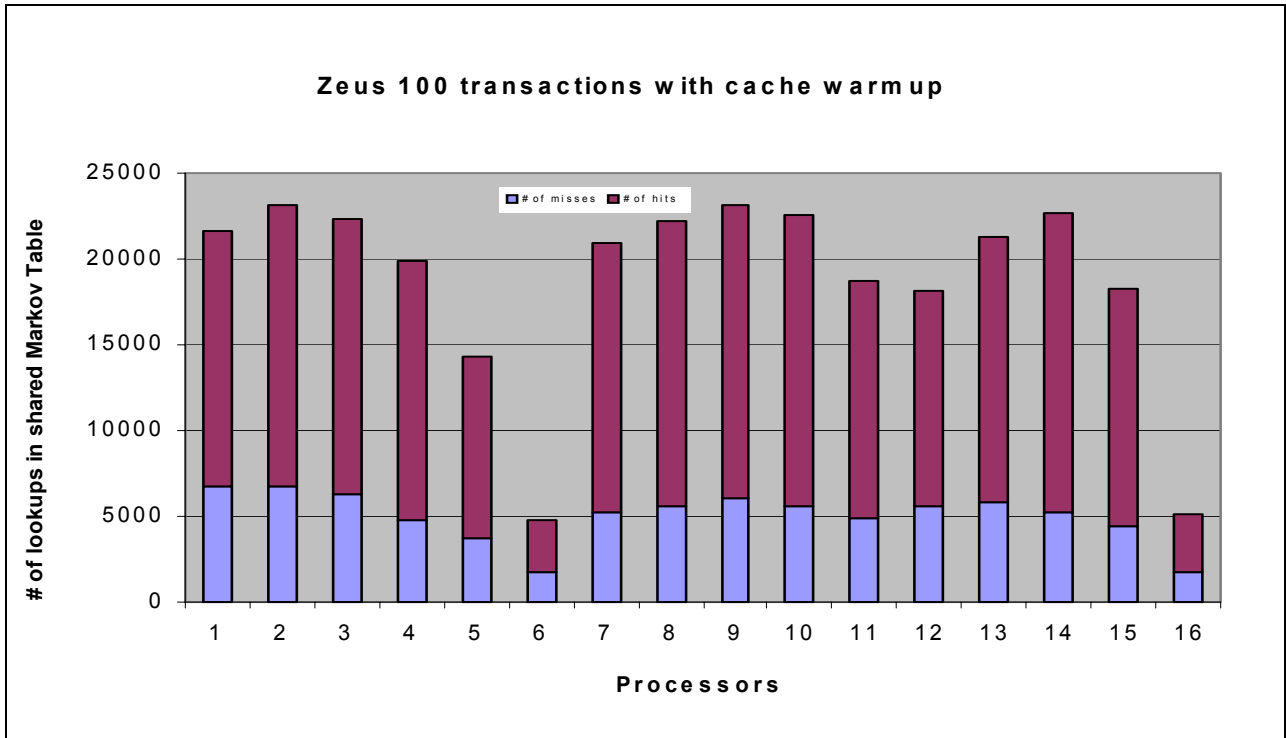
Figure 3 (a). Hit rate in the shared Markov table across 16 CMP cores for Zeus 100 transactions with cache warm-up.
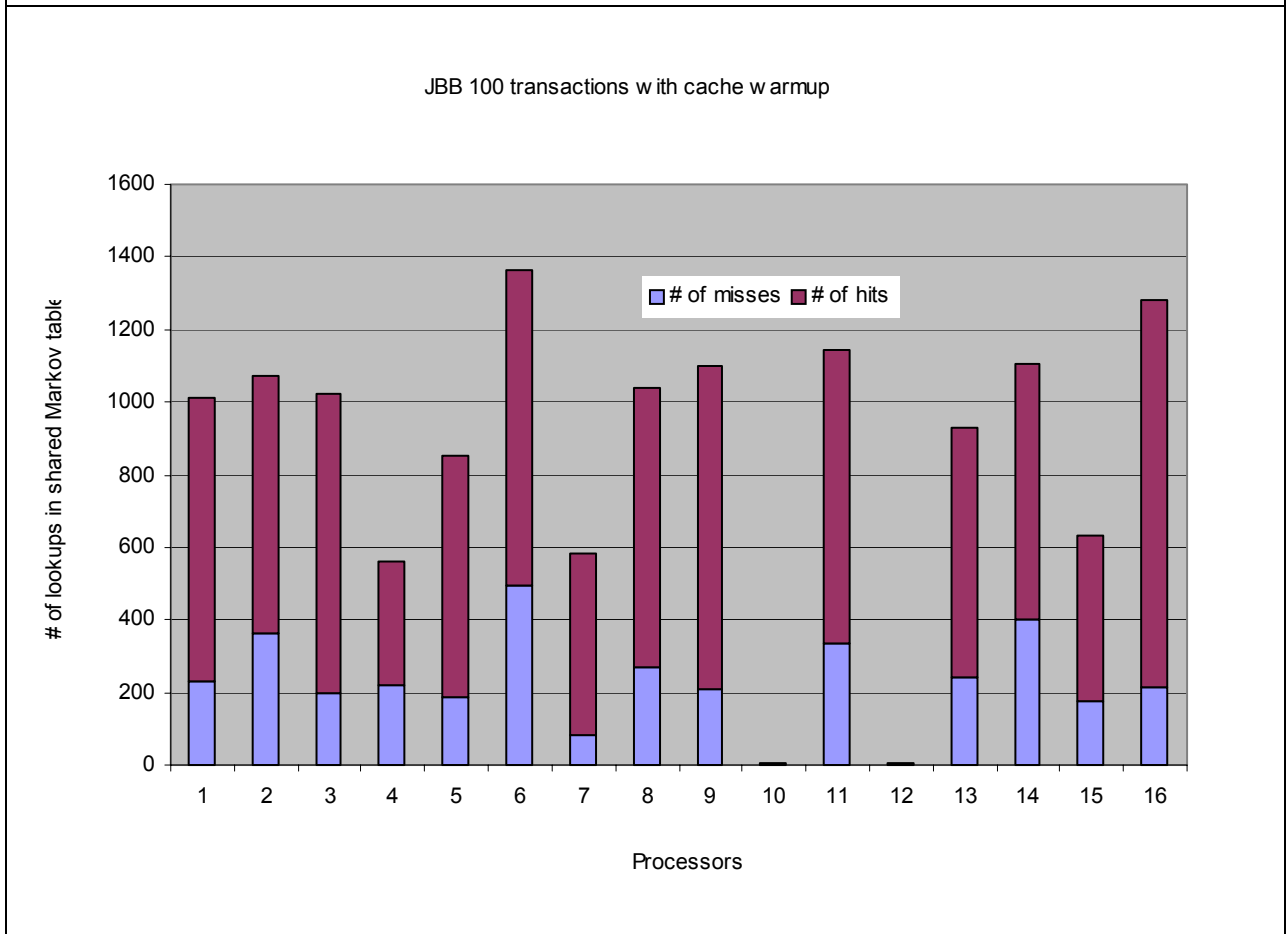
Figure 3 (b). Hit rate in the shared Markov table across 16 CMP cores for Jbb 100 transactions with cache warm-up.
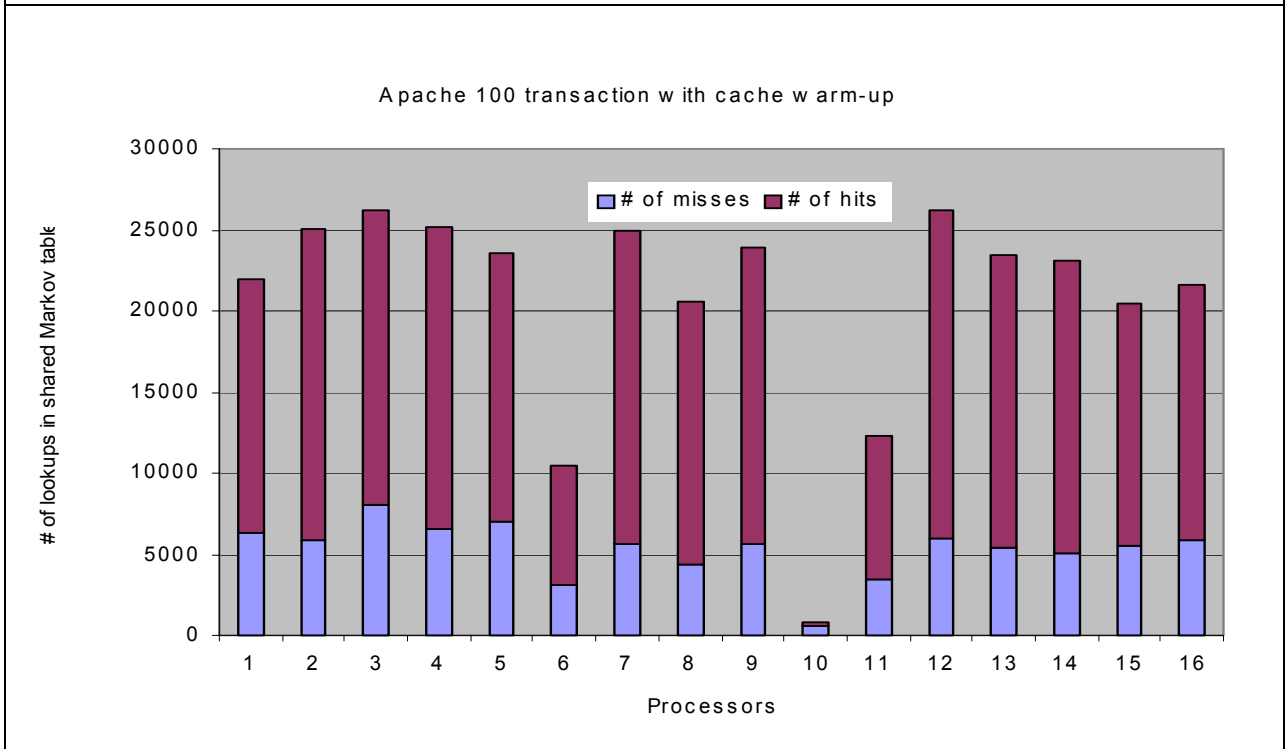
Apache 100 transaction with cache warm-up

Figure 3 (c). Hit rate in the shared Markov table across 16 CMP cores for Apache 100 transactions with cache warm-up.

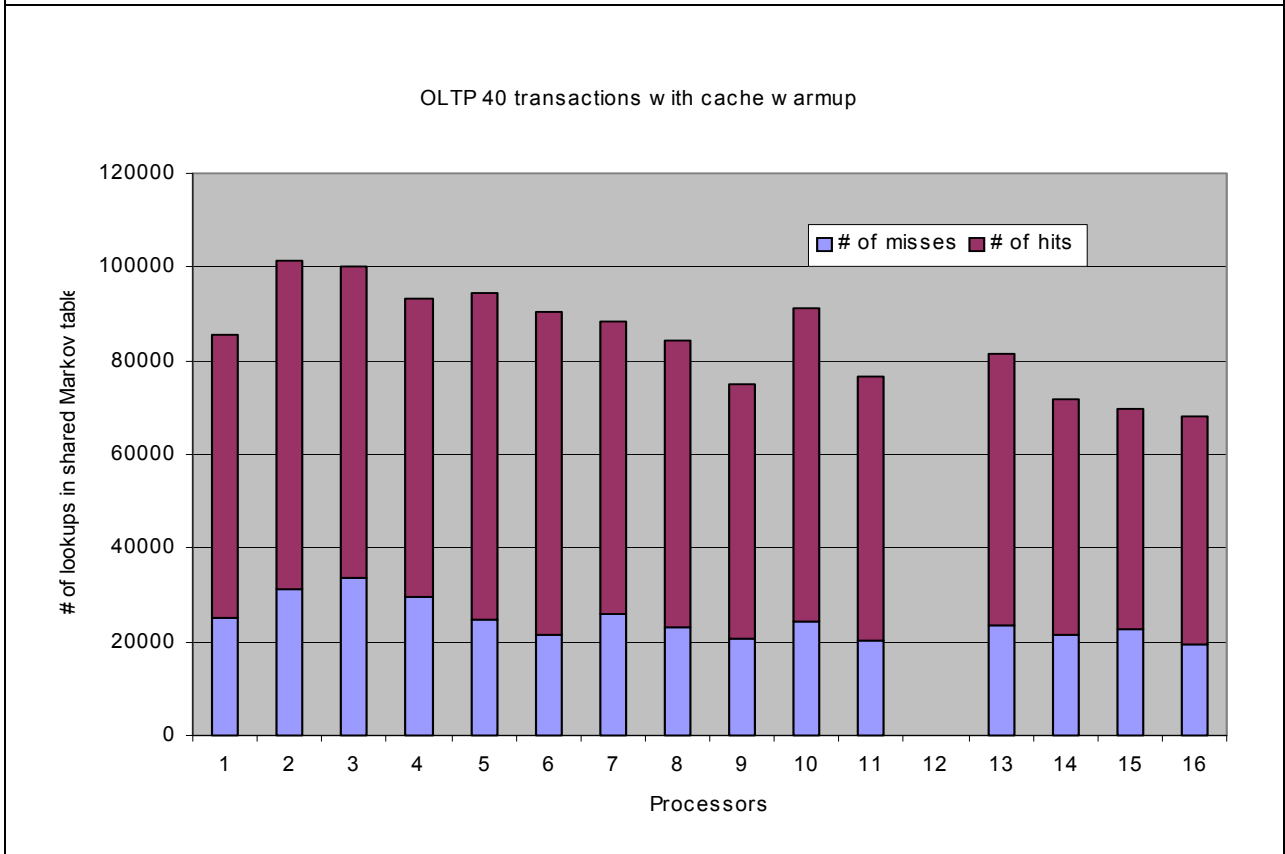**OLTP 40 transactions with cache warmup**

Figure 3 (d). Hit rate in the shared Markov table across 16 CMP cores for Oltp 40 transactions with cache warm-up

sor's.

We run our experiments over four benchmarks: Zeus, Jbb, Apache and Oltp. Due to the time constraint, we only run Zeus for 100 transactions, Jbb for 100 transactions, Apache for 100 transactions, and Oltp for 40 transactions. We notice that the load balance is fairly even for the four benchmarks except that one or two processors experience far fewer instruction cache misses than other peer processors. Among the four benchmarks, Jbb shows the worst load balance across different processors since Jbb has the least number of instruction cache miss rate. We believe that all the sixteen processors will show similar number of instruction cache misses if we run enough number of transactions for all four benchmarks.

Figure 4 Comparison of CMP and uP hit rates in the Markov table.
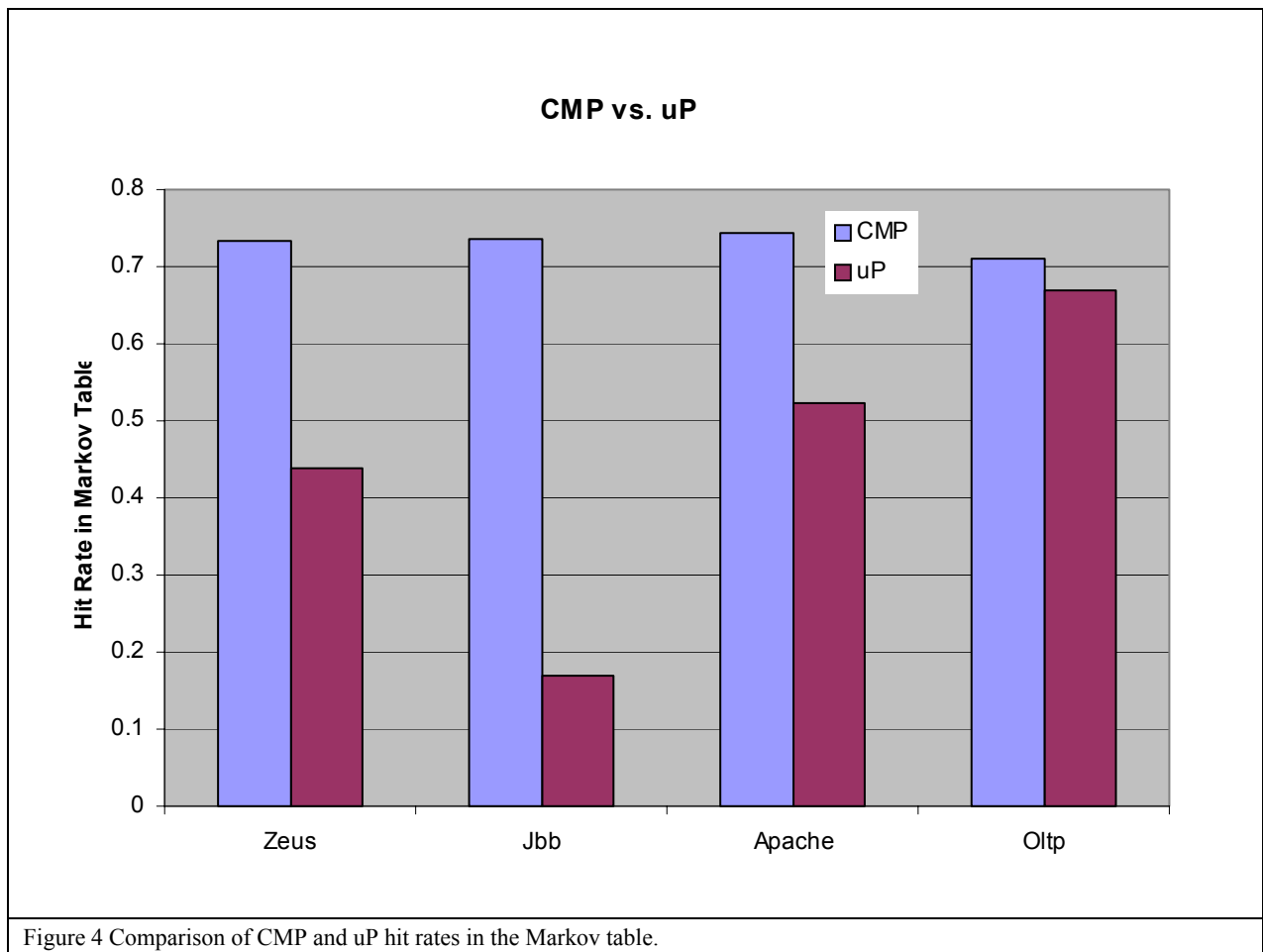
The four benchmarks exhibits very similar behavior in terms of the average hit rate and the core-specific hit rate. The average hit rate of all CMP cores for Zeus, Jbb, Apache and Oltp are 73.3%, 73,5%, 74.4% and 71.1%. Above 70% is a pretty good hit rate given the fact we only run 40 transactions for Oltp and 100 transactions for Zeus, Jbb, and Apache. We estimate that the hit rate may slightly improve if we run each benchmark for a few hundred of transactions. For each CMP core, its private hit rate always varies between 60% and 80% although some CMP core may experience more instruction cache misses than

their counterparts. There's only one exception here, which is processor 10 of Apache. We think this is probably the outcome of deviation. The hit rate of this processor should exceed 60% if we can run Oltp for several hundred transactions.

We further study the Markov table hit rate for uniprocessors and compare the CMP hit rate with uniprocessor hit rate to prove that the instruction sharing, esp. constructive interference exists in the instruction cache miss level on CMPs. We run 100 transactions for Zeus, Jbb, and Apache for uniprocessors as we did for CMPs. We run 40 transactions for Oltp for uniprocessors as we did for CMPs. See figure 4 for details. As we can see, the hit rates for Zeus, Jbb, and Apache dramatically increase from the uniprocessor case to the CMP case, which are typically more than 40%. The hit rate for Oltp increases by about 6% from uniprocessors to CMPs. There is still gain in terms of hit rate in the Markov table for Oltp even though the increase is not as dramatic as other benchmarks. Since Oltp is a data base transaction application, we think the same transaction might appear frequently on the same processor as well as on the other processors, which can explain why Oltp's hit rate increase from uniprocessors to CMPs is not as dramatic as the increases seen by the other three benchmarks. From this comparison, we can see that instruction sharing does exist in different cores' instruction cache miss streams and constructive interference helps to improve the Markov table hit rate from uniprocessors to CMPs.

6.4   Size of the Shared Markov Table

We have shown in section 6.4 that a shared Markov table can help reduce the number of instruction cache misses across CMP cores. Another problem worth studying is how big this table should be. In order to find the size of such a shared Markov table, we collect in our experiment the LRU distance of all the hit entries in the Markov table. The definition of the LRU distance was given before and it represents the chances that an entry is still in the table when it is looked for by a CMP core. More specifically, the larger the LRU distance of a hit entry is, the less possible it is that the entry is still in the table when it is used since this entry is closer to the LRU tail and may have been replaced.

As we can see in figure 5, all the hit entries are within 16K entries away from the LRU head entry since the table size is only 16K. Other than Jbb, about 60% hit entries are within 4K entries away from the LRU head entry, which indicates that a 4K-entry shared Markov table can still help reduce the number of instruction cache misses. A 4k-entry table is a very small table and actually a 16k-entry table isn't very big. From the circuit design perspective, a very small table like a 2k-entry table might cause trouble adding 4 or 8 read ports to itself. One thing worth pointing out here is that our 16K-entry table is a fully associative one which might be a little bit challenging to design. However, an 8K-entry fully associative table with significantly shorter access latencies than a big shared L2 cache can still be designed. In another approach, we

Figure 5. The number of hits that have LRU distances smaller than 16K, 4K, 2K and 1K.
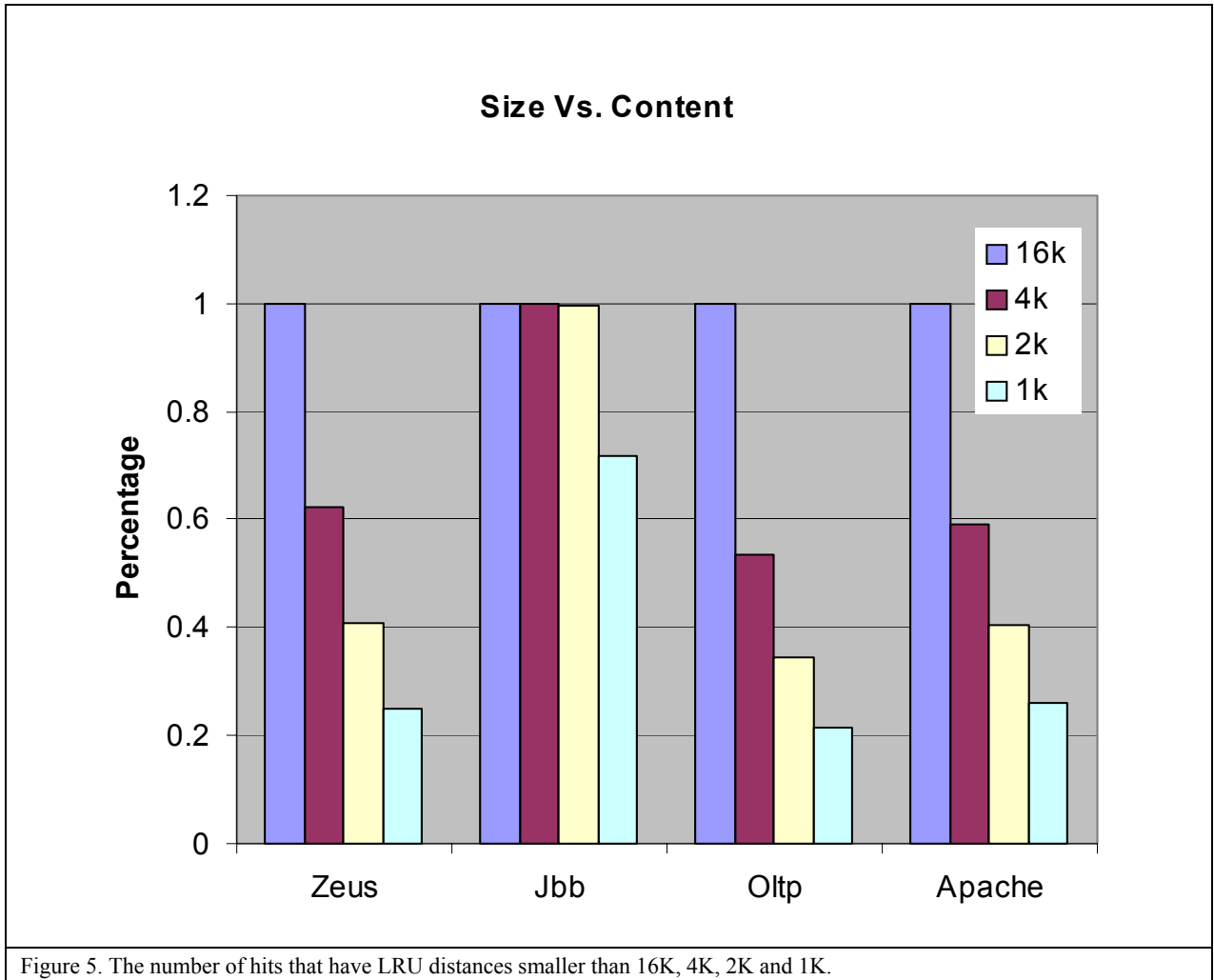
can still design a 32K-entry 8-way associative table or even a 64K-entry 8-way associative table.

6.5   Hardware Implementation Issues of a Shared Markov Table

As discussed in the previous section, we need to make decisions about the size and the associativity of a shared Markov table. We need to consider the area budget, access latency compared with a big shared L2 cache, and the pressure of many read/write ports added to the table. Another decision we can make is the number of CMP cores that can share such a table. For example, we can have two shared Markov tables and each 8 CMP cores instead of all the 16 cores share one single Markov table.

A simple variation of a shared Markov table is a shared history table for a Markov prefetcher. A CMP core starts prefetching instructions into its instruction cache when it experiences a cache miss and finds a hit in the shared Markov table. For a Markov history table, there might be several path that match one single instruction cache miss. Here we need to have a simple counter mechanism to choose from these several paths, the path that has the largest counter value.

Another hardware optimization for a shared Markov table is the separation of address directory

from data entries in the shared Markov table. A data entry, containing two cache lines, is much bigger than the sum of two cache block addresses. By the separation of data from addresses, we can afford having multiple copies of address directories within chip. We can have either one or two CMP cores sharing an address directory, which avoids crossing the whole chip for lookups and enables fast lookups in the table by CMP cores.

Another issue we want to study is the impact of sequential prefetchers on a shared Markov table. This problem can be addressed by examining the number of sequential cache misses for all two consecutive cache miss combination. If the number of sequential cache misses dominates, a sequential prefetcher might be a good idea instead of implementing a shared Markov table. If the number of sequential cache misses is small, it is definitely worth implementing a shared Markov table.

7 .0   Related Work

On-chip resource sharing has been extensively utilized for simultaneous-multithreading processors (SMTs). The original SMT processor[1] proposes to share functional units, first level instruction and data caches among different threads within one single processor core. Intel's patented hyperthreading technology [2] shares memory hierarchy, a few pipeline stages (register renaming, instruction issuing, execution, memory, register writeback) between two simultaneous threads. However, researchers have pointed out that sharing should not be excessively implemented for SMT processors. Matt et. al. [3] identified that sharing branch predictor history among different threads could hurt the overall performance for SMT processors by causing destructive interference.

On-going CMP research has started studying sharing on-chip resources across different CMP cores. IBM power 4 [4] shares an L2 cache between two on-chip cores. Academic CMP research [5] also proposes sharing second-level caches among different cores.

Other microarchitecture study, CMP research also started investigating server-side application characteristics, e.g., OLTP [6], ECPerf [7],

8 .0   Conclusions and Future Work

In conclusion, we have made several contributions to ongoing CMP research in our project. First, we find that the instruction-based sharing exists for current commercial workloads running on multiple CMP cores. Constructive interference also exists and can be explored. Second, we propose a small Markov table underneath L1 instruction caches and shared by all CMP cores. We demonstrate that this small table can help reduce instruction cache misses and can potentially improve the overall CMP performance. Third, we have explored the design space of the newly proposed Markov table. We show that the size of this table can vary from 4K to 32K to efficiently reduce instruction cache misses. We also identify several design

optimizations and investigate the interference of the Markov table with other on-chip hardware structures.

For future work, we need to perform experiments to get the IPC data and see how much performance improvement a shared Markov table can bring. We also need to investigate the interference between a shared Markov table and a sequential prefetcher. We need to find how many consecutive instruction cache misses are sequential. Third, we need to study the possibility of implementing Markov prefetching based on the existing instruction-based sharing. Finally, we need to find out if the data-based sharing exists on CMPs and if the larger thread-level sharing exists on CMPs.

References

[1]   Dean Tullsen et. al. Simultaneous Multithreading: Maxmizing On-Chip Parallelism. In proceedings of the 22th Annual International Symposium on Computer Architecture (ISCA-22), June, 1995

[2]   Deborah Marr et. al. Hyper-Threading Technology Architecture and Microarchitecture. Intel Technology Journal, Issue 01, Volume 06, Feburary, 2002.

[3]   Matt Ramsay et. al. Exploring Efficient SMT Branch Predictor Design. Workshop on Complexity-Effective Design, in conjunction with ISCA, June, 2003.

[4]   J. M. Tendler et. al. POWER4 System Microarchitecture. IBM Systems Journal, Volume 46, NO. 1, 2002.

[5]   Kunle Olukotun et. al. The Case for a Single-Chip Multiprocessor. Proceedings of the 7th Intenational Symposium on Architectural Support for Parallel Languages and Operating Systems, October 1996.

[6]   Luiz Andre Barroso et. al. Impact of Chip-Level Integration on Performance of OLTP Workloads. Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA-6), January 2000.

[7]   Martin Karlsson et. al. Memory System Behavior of Java-Based Middleware. Proceedings of 9th International Symposium on High Performance Computer Architecture (HPCA-9), February 2003.

[8]   Y.N.Patt et. al. HPS, a New Microarchitecture: Rationale and Introduction. Proceedings of 18th Annual Workshop on Microprogramming, December, 1985.

[9]   Joseph Fisher Very Long Instruction Word Architectures and the ELI-512. Proceedings of the International Symposium on Computer Architecture, 1983

[10]   http://www.intel.com/pressroom/kits/events/enterprise_server/itanium_ecosystem.pdf. The Intel Itanium Architecture Comes of Age

[11]   J.DWarnock et. al. The Circuit and Physical Design of the POWER4 Microprocessor. IBM J. of Research and Development, Vol. 46, No.1, 2002.

[12]   G.S.Sohi et. al. Multiscalar Processors. Proceedings of 22th International Symposium on Computer Architecture, 1995.

[13]   Dean Tullsen, et. al. Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor. Proceedings of 5th International Symposium on High Performance Computer Architecture, January, 1999.

[14]   J.Huh et. al. Exploring the Design Space of Future CMPs. Proceedings of International Conference on Parallel Architectures and Compilation Technologies (PACT), September, 2001.

[15   Alaa Alameldeen et. al. Simulating a $2M Commercial Server on a $2K PC. IEEE Computer, February, 2003