

A Case for Shared Instruction Cache on Chip Multiprocessors running OLTP

Partha Kundu, Murali Annavaram, Trung Diep, John Shen
Microprocessor Research Labs, Intel Corporation
{partha.kundu, murali.annavaram, trung.diep, john.shen}@intel.com

Abstract

Due to their large code footprint, OLTP workloads suffer from significant I-cache miss rates on contemporary microprocessors. This paper analyzes the I-stream behavior of an OLTP workload, called the Oracle Database Benchmark (ODB), on Chip-Multiprocessors (CMP). Our results show that, although, the overall code footprint of ODB is large, multiple ODB threads running concurrently on multiple processors tend to access common code segments frequently, thus exhibiting significant constructive sharing. In fact, in a CMP system, an I-cache shared between multiple processors incurs similar miss rate as a dedicated I-cache per processor where the per processor I-cache has the same capacity as the shared I-cache. Based on these observations, this paper makes the case for a shared I-cache organization in a CMP, instead of the traditional approach of using a dedicated I-cache per processor.

Furthermore, this paper shows that OLTP code stream exhibits good spatial locality. Adding a simple dedicated Line Buffer per processor can exploit this spatial locality effectively, to reduce latency and bandwidth requirements on the shared cache. The proposed shared I-cache organization results in an improvement of at least 5X in miss rate over a dedicated cache organization, for the same total capacity.

1. Introduction

With silicon technologies on the threshold of a billion transistors, the era of Chip Multiprocessor (CMP) has already arrived [1][2]. Current CMPs allow a modest number of processors (2-8 simple processor cores) to be integrated along with a relatively large cache memory on a single silicon die.

Barroso *et al.* [1] proposed Piranha, a research prototype CMP that integrates eight simple Alpha processor cores along with a two-level cache hierarchy onto a single chip. Piranha was able to outperform its next-generation processors by up to 5 times on OLTP workloads. Piranha however, primarily focused on improving the D-cache performance and did not analyze design alternatives for the I-cache.

Ranganathan *et al.* [6] used trace-driven simulations to show that a perfect instruction supply would yield 30% better performance, over the combined effects of a perfect branch predictor, very large instruction window and infinite functional units. Ailamaki *et al.* [3] analyzed three commercial DBMSs on a Xeon processor and showed that TPC-D queries spend about 20% of their execution time on L1 instruction cache miss stalls.

Given the significant performance degradation due to instruction supply bottleneck, we believe that improving instruction cache behavior of OLTP is an important first step toward designing an efficient CMP for OLTP in the future.

This paper makes three main contributions. First, it characterizes the instruction stream of one of the most commonly deployed enterprise class databases in the industry - the Oracle database configured to run an OLTP set-up. Second, it shows that although the I-cache footprint of the Oracle Database benchmark (ODB) is large, multiple ODB threads tend to access common functions frequently, thus exhibiting significant constructive behavior. Third, the paper proposes a prototype I-cache organization that departs from the traditional approach of using per processor first-level caches in a CMP; replacing individual I-caches by a large shared I-cache, our results show over 5X improvement in miss rates per processor while maintaining the same latency as a dedicated I-cache.

The rest of the paper is organized as follows: Section 2 explains our workload tuning and system level simulation methodology. Section 3 presents characterization of the OLTP workload. Section 4 presents the baseline cache organization, against which improvements in the paper are compared. In section 5 we examine the effects of multiprocessing on a shared instruction cache. In section 6 we characterize the effect of placing a small buffer (called a Line Buffer) close to a processor core. Section 7 describes the new cache organization and presents the gains achieved from the proposed cache organization. Section 8 describes relevant prior work and we conclude in section 9.

2. Workload and Simulation Environment

2.1 Benchmark Description

In this study, we used an Oracle 8.1.6 based OLTP workload, which we call the Oracle Database Benchmark. ODB simulates an order-entry business system, where terminal operators (or clients) execute transactions against a database. The database is made up of a number of warehouses. Each warehouse supplies items to ten sales districts, and each district serves three thousand customers. Typical transactions include entering and delivering customer orders, recording payments received from customer, checking status of a previously placed order, and query the system to check inventory levels at a warehouse.

When ODB starts execution, it spawns three types of processes: user processes, server processes and background processes. A user process executes client's application code, such as parsing a query, and submits an appropriate data access request to a server process. A server process accesses the database on behalf of a user and provides user requested data. More than 90% of ODB execution time is spent in server processes. Background processes perform database maintenance tasks such as log management, committing modified data to disk, and managing locks amongst competing server processes. All Oracle processes, server as well as background processes, share a large memory segment called the System Global Area (SGA). A large portion of SGA is devoted to the database buffer cache, which holds the working set of a database in memory. The database buffer cache tracks the usage of the database blocks to keep the most recently and frequently used blocks in memory, significantly reducing the need for disk I/O.

2.2 Setup and Validation of ODB

Commercial workloads, such as ODB, need careful tuning to reduce I/O wait time. A production ODB run typically uses hundreds of warehouses and requires hundreds of gigabytes of disk space and tens of gigabytes of physical memory. Such a setup is typically not amenable for detailed system level simulation studies. Hence, in this study we use an in-memory ODB setup, where the working set fits in memory with negligible amount of disk I/O.

To achieve the goal of running ODB completely in-memory we tune the setup parameters; increasing the SGA size, reducing the number of warehouses to reduce the working set size, and using enough clients for optimal concurrency. Tuning requires repeated execution of user transactions after modifying each setup parameter and even building the database from scratch multiple times.

Since these operations are very time consuming, we first develop and test ODB on a native machine before running on a simulator.

We use a Pentium III system running Linux Red Hat 7.2 with 2 GB main memory and 70 GB disk space. We use a 10-warehouse setup, which occupies 35 GB of disk space. We configured the Oracle server to use 1.5 GB of memory for SGA, which is sufficient to cache frequently accessed database tables and metadata in main memory. Nevertheless, some disk I/O does occur; mostly non-critical writes of the redo log buffers. Our tuned version of ODB has less than 2% idle loop overhead. After tuning ODB, we built a complete and exact disk image of the ODB workload that can then be executed by the Simics [13] simulator.

2.3 Tracing Using Full System Simulator

We use Simics, a full system simulator, to simulate the tuned ODB workload identical to that run on the native hardware. Figure 1 graphically depicts the various steps involved in our tuning and simulation process. Simics is a full system simulator that is capable of booting several unmodified commercial operating systems and running unmodified application binaries.

We configured Simics to model a Pentium III processor running Red Hat Linux 7.1 (to use the Linux version available for Simics, that most closely matched our native set-up). We built a disk image of the tuned ODB for loading into Simics. We simulated ODB on Simics for a sufficiently long time to warm up the main memory buffer cache in the simulator. We use Simics checkpoint capability to store a snapshot of the memory and disk state after warming up the database. Checkpointing allows us to start the simulation from exactly the same state each time we run our simulation. We use these checkpoints for collecting traces used in this study.

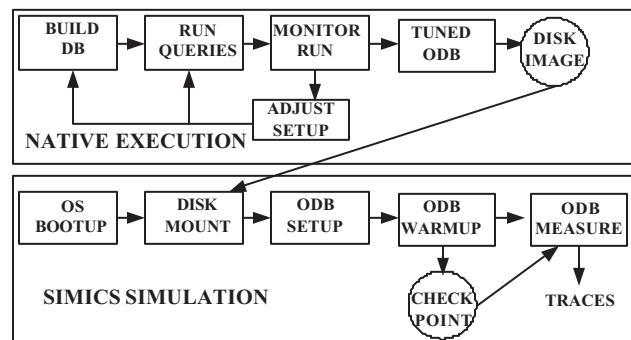


Figure 1: Tuning and Simulation Methodology

Simics provides a default trace module that is capable of generating instruction and data traces. We modified the

trace module to generate information such as virtual and physical addresses of instruction and data, instruction type (e.g. branch instruction, memory instruction), instruction length, instruction assembly, and process ID of the process that generated a trace record. We run ODB for one billion instructions from the checkpoint and collect instruction and data traces that were then analyzed using a cache simulator (developed in-house).

3. ODB Instruction Cache Characteristics

In the first part of this section, we present some profile data for an ODB trace. In the second part, we present characterization data for a range of I-cache parameters. The data presented in this section pertains to a uni-processor (1P) system.

3.1 ODB Code Footprint Profile

In order to obtain an execution profile of the ODB code stream, the ODB code references were grouped into unique chunks of 64 bytes each. We then collected statistics on the execution frequency of these chunks. Figure 2 shows the size of I-cache required to fit a given fraction (%) of the ODB footprint.

The data shows that the I-cache size requirement grows exponentially, relative to the fraction of code executed. For instance, the graph shows that less than 27 KB of space is needed to capture 50% of the dynamic instruction reference stream while more than 330 KB of space is needed to capture 95% of the reference stream. Figure 3 captures the average number of instructions executed in between repeated references to the same code chunk. This chart provides an indication of the temporal locality that exists for the various fractions of the footprint. The data shows that 50% of the code chunks are re-executed within 2000 instructions.

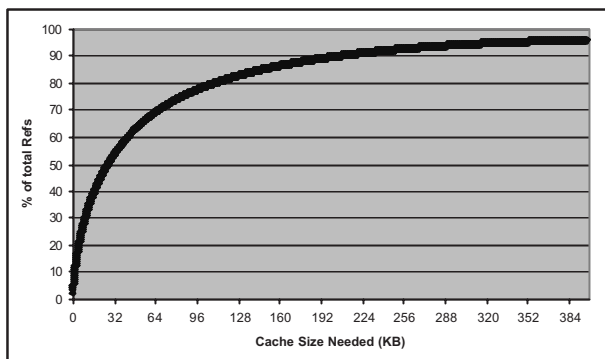


Figure 2: I-Cache Size Requirement to cover ODB footprint

This leads us to conclude that although the ODB code footprint is significantly large, more than 50% of the references can fit within a modest 32 KB cache (provided no capacity or conflict misses are inflicted by the remaining code chunks). Furthermore, ODB code has good temporal locality due to code reuse.

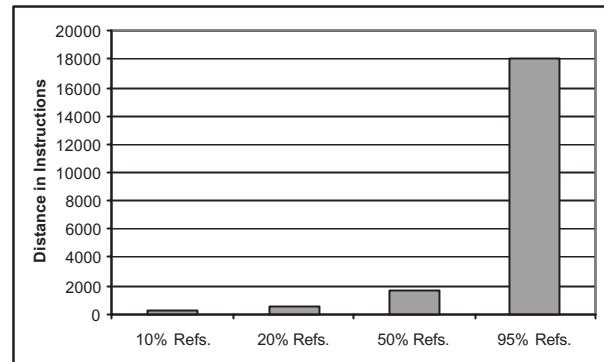


Figure 3: Avg. Distance Before a Repeat Access to a Chunk

3.2 Impact of I-cache

In this section, we look at cache behavior when running the ODB code. Figure 4 shows the miss rates (misses per 1000 instructions) for a range of cache capacities. As shown, miss rates decrease steadily, until about 512 KB, which represents the 99-percentile point of the full ODB dynamic trace. These results show that the overall code footprint of ODB is significantly larger than the typical I-cache sizes seen in current processors.

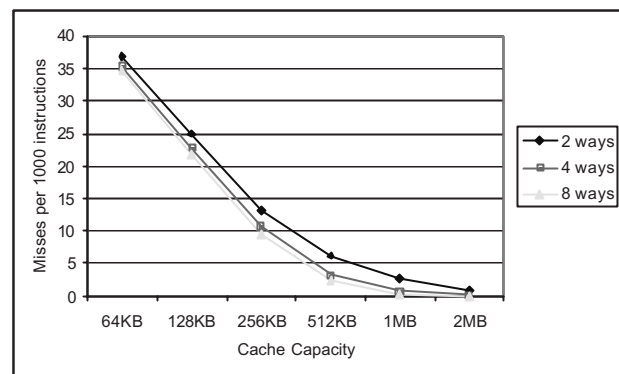


Figure 4: Effect of I-cache Size on Miss Rate

Figure 4 plots the effect of associativity on the miss rate. At the range of capacities typically used in the first-level I-cache (32-64KB), we see that increasing the associativity of the cache has little or no effect on miss rates. Capacity misses dominate the overall miss rate, thus making the conflict misses a negligible fraction of the total misses. However, when using larger cache sizes, miss rates drop by nearly 50% as the associativity is increased

from 2 to 8. These results are consistent with those reported by Barroso *et al* [5].

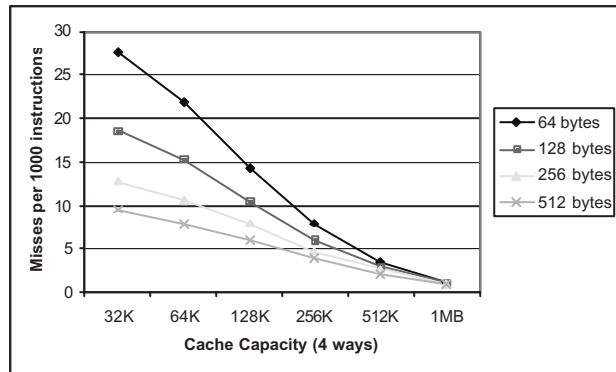


Figure 5: Effect of Line Size on Miss Rate

Figure 5 shows the effect of increasing line size on miss rates. At lower capacities, the effect of increasing the line size is more pronounced than at larger capacities. For the relatively small cache capacities typically employed at the first-level I-cache, we see that increasing the line size may reduce miss rates, quite significantly.

4. Description of Baseline Architecture

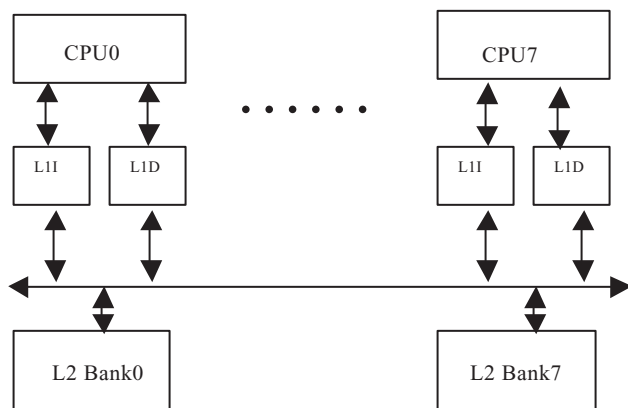


Figure 6: Baseline Cache Organization

Figure 6 shows the on-chip cache organization as described in [1]. The memory controller and packet router are not shown. There are eight processors (CPU0-7) on the chip. Each CPU has its own dedicated I- and D-cache. The first-level caches are connected to each other and to a banked second-level unified cache through a high bandwidth, intra-chip switch.

The first-level caches are each 64KB, 2 way set associative with 64-byte line sizes. This design will be referred to as the baseline throughout this paper.

5. Shared Cache for CMP

As shown by Barroso *et al.* [1], a chip multiprocessor (CMP) is particularly attractive for OLTP workloads. In previously proposed CMP designs, however, the first-level I- (and D-) caches are replicated for each CPU thereby distributing the allocated cache budget amongst several processors. For instance, for the baseline 8-processor CMP, even when the net I-cache capacity is 512 KB, each processor only sees the benefits of its own 64 KB cache.

Motivated by the observation that ODB has a large code footprint, we explore the effects of combining the dedicated caches of each CPU to form a larger shared cache.

For an 8-processor CMP, we choose two configurations for study:

- I. 4 clusters with 2-CPU's per cluster (Shared 2P)
- II. 2 clusters with 4-CPU's per cluster (Shared 4P)

Each cluster has an I-cache that is shared by the CPUs in that cluster. For all the CMP studies, we started with a single instruction trace collected from our ODB set-up running on a simulated 8-way SMP system. A 2-P and a 4-P cluster is studied by post-processing the instruction trace records from two and four of the eight processors in the trace, respectively. Using a single trace to model different CMP configurations reduces the variability that often results from examining different MP setups [10].

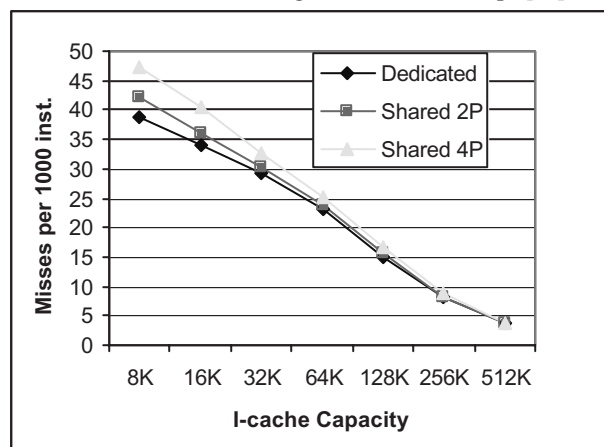


Figure 7: Effect of combining dedicated caches on a CMP

Figure 7 shows the miss rates for each of the two configurations (Shared 2P and Shared 4P) contrasted against the miss rate of a dedicated cache. The data shows that a 256KB cache, shared between 4 processors has a miss rate that is within 6% of that achieved by a dedicated (per processor) I-cache of 256KB.

Furthermore, compared with the baseline I-cache of 64KB dedicated per CPU, we obtain about 33%

improvement in miss rate, if the consolidated capacity (of 128KB) is shared between 2 processors or 62% improvement when the net capacity of 256KB is shared between 4 processors.

5.1 Constructive vs. Destructive effects in a Shared Cache

In order to understand the beneficial effects of a shared cache, we focus on two effects:

1. *Capacity Increase*: Cache blocks that are shared between multiple processors avoid the un-necessary replication that occurs on a dedicated cache. The cache area thus saved can potentially increase the effective capacity seen by a processor.

2. *Pre-fetching*: When a block that is demand fetched by one processor also ends up being used by another processor, then the first processor is said to have pre-fetched the block for the second processor. In the ideal case, if the program phases of two processors were to line up exactly, one processor will always pre-fetch for another processor – leading to a halving of the miss rate.

The effect of *capacity increase* is measured by counting the fraction of blocks in the cache that are shared by two or more processors.

The effect of *pre-fetching* of cache blocks by one processor for other processor(s) is measured using a sharing mask per cache block (a bit in this mask represents a processor that used this cache block since it was brought into the cache). A block is considered pre-fetched, if a processor attempting to access the cache block finds, that it has been fetched by another processor (i.e. the processor has not accessed this block since it was allocated into the cache). As such, these references are likely to be misses in a dedicated cache of this processor.

To counter-act the above beneficial effects of a shared cache, references from multiple processors may conflict on the cache index and thus replace those blocks that are being used by another processor. We term these as *destructive misses*. We measure *destructive misses* by counting the number of blocks in the cache that are evicted by a processor not using the block, before being brought back into the cache by one of the processors that was using the block (prior to the eviction).

In Table 1 we present the data for a 256KB cache shared between 2 processors (2P) and 4 processors (4P). The first column (% blocks shared) presents the measure of *capacity increase* as described above. Thus, for each of the 2P and 4P cases, close to half of the cache is shared between processors. Similarly, the column (Blocks pre-fetched) shows the measure of *pre-fetching* by one processor for another (measured as a fraction of total instructions). Finally, the last column shows the effect of

destructive interference by one processor on other(s). Later in Section 6.2 we will see that by reducing the destructive misses on a shared cache, we cause a dramatic improvement in the *pre-fetching* as well as the *capacity increase* metrics .

Table 1: Constructive & Destructive Effects of a 256KB Shared Cache

% blocks shared		Blocks pre-fetched (per 1000 instructions)		Destructive Misses (per 1000 instructions)	
2P	4P	2P	4P	2P	4P
42.12%	54.33%	2.37	5.39	10.17	10.78

5.2 Clustering of functions in ODB

In this section, we will attempt to develop an intuition into the way different threads in ODB share their code stream. We refer back to 3.1, which classified references as being the top 10%, 20% and 50% based on the frequency of their execution. We see that these references are called on an average, approximately every 100, 700 and 2000 instructions, respectively. Thus, the functions in the top 10% have better temporal locality than the functions in the top 20% and so on.

Table 2: Periodicity of ODB Functions

MP configuration	Avg. Cycles before Functions Re-executed (in instructions)		
	10%	20%	50%
1P	262	528	1702
2P	284	544	1749
4P	320	701	2569

As shown in Table 2 multiple Oracle server threads operating concurrently (2P and 4P) exhibit similar temporal locality as a single thread running alone.

Therefore, we see that multiple threads spend a good deal of time executing some common sections of code. Such references are well served by a shared cache.

Conversely, when threads execute paths that do not intersect in time with other threads, then these references tend to conflict and create misses that we term destructive.

6. A Line Buffer to replace the dedicated I-cache

Section 5 shows the benefits of a shared instruction cache over a dedicated cache scheme for ODB. There are,

however, a number of engineering obstacles to designing a shared I-cache. A single shared cache servicing multiple CPUs would be expected to deliver a much higher bandwidth than a dedicated cache serving a single CPU. For instance, replacing four dedicated caches with one large shared cache would necessitate a four-fold increase in request traffic to the shared cache. Furthermore, a larger shared cache would have longer access latency than a smaller dedicated cache, which may stall the processor pipeline, thereby negating the effects of a lower miss rate. Increasing the cache size four-fold from 64KB to 256KB would increase the latency of the cache by almost 50% [8].

In order to alleviate the bandwidth and latency constraints, we propose adding a per-processor Line Buffer [8] to the shared cache. In the following Section, we show that by aggressively pre-fetching 4-8 blocks in advance and storing these fetched blocks in a Line Buffer, we achieve a miss rate equal to that of the baseline I-cache.

6.1 Capacity vs. Bandwidth

We see in Section 3.2 that the miss rate of the ODB code stream decreases steadily as the line size of the cache is increased (at the lower capacities). Increasing the line size or alternately, doing next-line pre-fetching [7], have been proposed in the past to reduce the miss rate in I-caches.

Next-line pre-fetching or streaming, however, comes at a price – it costs more in terms of data bandwidth to transfer the additional bytes from the shared cache to the processor. As shown in Figure 8 the number of instructions executed per extra byte fetched, decreases as a square function of line size, illustrating the decreasing benefit of line size increases.

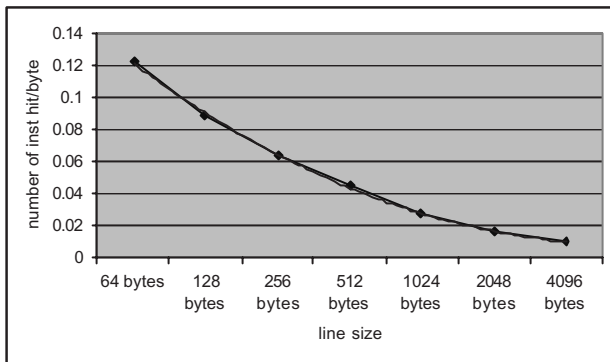


Figure 8: Instruction Yield per extra byte pre-fetched

Nevertheless, across the spectrum of line sizes studied, the number of valid instructions found keep increasing as the line size is increased.

Figure 9 provides a relative comparison of the miss rates of a few different Line Buffer configurations. We see that an 8 entry Line Buffer that fetches 256 bytes at a time (2KB) and a 2 entry Line Buffer that fetches 512 bytes at a time (1KB), both have miss rates very similar to the baseline I-cache (64KB). However, the 1KB Line Buffer uses twice as much data bandwidth as the 2KB, which in turn, uses four times as much bandwidth as the baseline cache of 64KB (with 64byte line size).

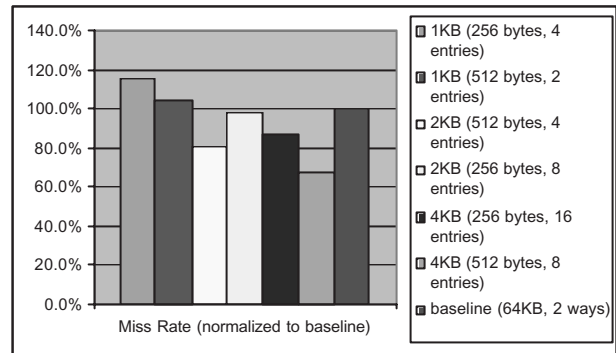


Figure 9: Miss Rates of various Line Buffer configurations normalized to Baseline

We therefore note that for the ODB workload, bandwidth may be traded off against cache capacity for a specific miss rate. Bandwidth on a chip is more readily obtained (compared to SMPs constructed from commodity parts) by increasing the toggle rate on wires. Therefore, it seems prudent to invest the allocated cache transistors towards a shared cache while building wiring interconnects to deliver large bandwidths between the shared cache and the processors.

Thus, a Line Buffer can be constructed to effectively manage the request bandwidth into the shared cache while keeping the access latency within tolerable limits.

6.2 Effect of Line Buffer on Shared Cache

Table 3 presents the metrics (*capacity increase*, *pre-fetching* and *destructive misses*) that we defined in Section 5.1, for a 256KB shared cache. We see that the presence of the Line Buffer decreases the number of *destructive misses* by almost 3-fold, in each of the 2P and 4P cases, over that shown in Table 1.

As less of the blocks, fetched in a given time, are replaced by conflicting accesses from other processors, they live in the cache long enough, to allow other processors, executing a similar path, to use them – this is

shown in the improvement in the *pre-fetching* as well as the *capacity increase* metric (Table 3).

Table 3: Effect of Line Buffer on a 256KB Shared Cache

% blocks shared		Blocks pre-fetched (per 1K instructions)		Destructive Misses (per 1K instructions)	
2P	4P	2P	4P	2P	4P
73.18%	82.53%	21.59	49.51	2.88	3.10

7. Proposed Cache Organization

Figure 10 shows a 4P cluster (within the 8P CMP) illustrating a shared I-cache. Each processor first accesses the Line Buffer. If the Line Buffer misses, the access is sent to the shared cache. Accesses from the four processors arbitrate for the shared cache.

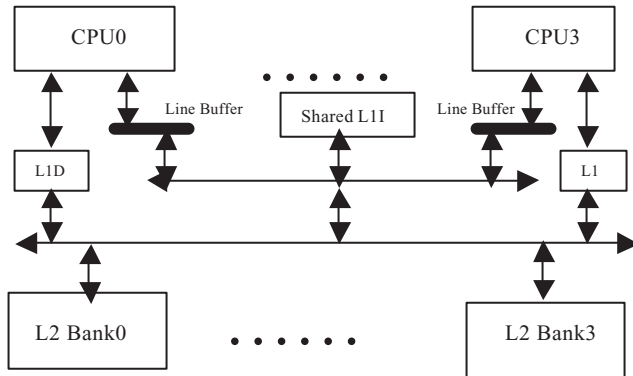


Figure 10: Proposed Cache Organization

In the event of a miss in the shared cache, the request is sent to the large L2 cache. The rest of the memory hierarchy remains similar to a traditional CMP design.

7.1 Miss Rate of Proposed Cache

In section 5 we saw that the miss rate of the shared cache was similar to that of a dedicated cache of the same capacity. Figure 11 compares the improvement in the effective miss rates for each of the 2P and 4P configurations when used with a Line Buffer and shared cache (shown in Figure 10). With a Line Buffer interfacing to the shared cache, we achieve a 30% improvement in miss rate over a dedicated cache of the same size (baseline w/256KB). The miss rate improvement over the baseline (of 64KB dedicated cache with 64 byte lines) is over 8-fold while compared to a baseline (64KB) with a line size of 256 bytes, we see an improvement of 5X.

Thus, not only does the cache organization of Figure 10 improve the miss rate significantly over the baseline, it also improves upon the shared cache described in section 5.

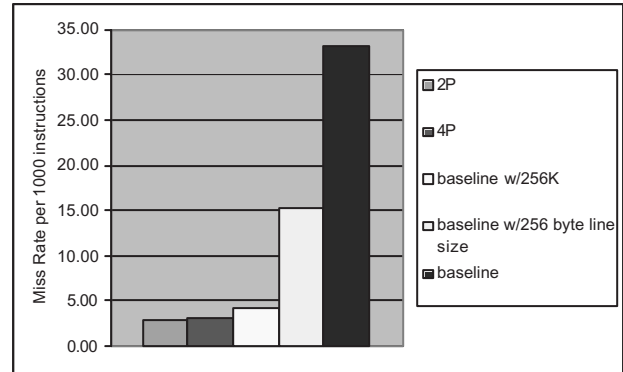


Figure 11: Miss Rate Comparison

8. Related Work

In the domain of chip multi-processing, several papers [2] have advocated CMP in the context of SPEC [9].

Of late, there have been a number of studies on database applications due to the increasing importance of these workloads [5][6]. Barroso *et al* [1] first proposed the idea of using a CMP for the express purpose of multiplying OLTP performance on a single chip. To the best of our knowledge, there has been no other published work to improve over the gains reported since then.

Simultaneous multithreading (SMT) is an alternative to CMP for exploiting the thread-level parallelism in commercial workloads. In fact, Lo *et al* [11] have shown that SMT can provide a substantial gain for OLTP workloads, although SMT processors add extra resources (e.g. larger register file) and design complexity to support multiple simultaneous threads.

Ramirez *et al* [12] offer code layout optimizations to improve instruction cache performance on OLTP. They report about 50% improvement through basic block chaining, fine-grain procedure splitting and procedure ordering. The greatest gains reported were through basic-block chaining. We find that these techniques to improve miss rates for a single processor system interact favorably in the context of a shared cache proposal presented in this paper. For instance, procedure ordering and basic-block chaining both improve spatial locality which aids the Line Buffer. Procedure-ordering also helps to cluster functions in the cache, which in turn improve temporal locality across threads.

9. Discussion and Concluding Remarks

As has been shown in previous work, chip multiprocessors are well suited for OLTP workloads. However, there remain several bottlenecks. This paper deals with one such bottleneck: the instruction code footprint. Examining the Oracle database benchmark, the paper observes that it would take 4-8X the size of a typical I-cache to capture 95% of the footprint. There are obvious impediments to providing this level of I-cache: processors, despite having a larger and larger transistor budget, can never approach the cache capacity of off-chip caches. Furthermore, a CMP with several processor cores can quickly deplete the transistor budget, if they are each given a large cache to feed their processing needs.

This paper makes the case to divide future CMPs into small clusters. Each cluster of 2 or 4 processors, with a shared instruction cache. The paper shows that the constructive and destructive interactions of the 2 or 4 ODB threads running concurrently on this shared cache, mostly balance out. So, that in the net, each processor virtually sees the benefit of the full larger cache. Furthermore, by adding a very small structure - the Line Buffer - to buffer the last few contiguous cache blocks, the paper shows a multiplying effect on the hit rate of the shared cache. The paper establishes that the Line Buffer drastically reduces destructive interference at the shared cache and provides at least 5X improvement in overall miss rate, for the same total cache capacity.

10. References

[1] L.A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzky, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 282-293, June 2000.

[2] K. Olukotun, B.A. Nayfeh, L. Hammond, K. Wilson and K. Chang. The Case for a Single-Chip Multiprocessor. In *Proceedings of the 7th International Symposium on Architectural Support for Parallel Languages and Operating Systems*, pages 2-11, October 1996.

[3] A. Ailamaki, D. DeWitt, M. Hill, and D. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 266-277, September 1999.

[4] K. Keeton, D.A. Patterson, Y.Q. He, R.C. Raphael, and W.E. Baker. Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 15-26, June 1998.

[5] L.A. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 3-14, June 1998.

[6] P. Ranganathan and K. Gharachorloo and S.V. Adve and L.A. Barroso. Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 307-318, October 1998.

[7] N.P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364-373, May 1990.

[8] K.M. Wilson and K. Olukotun. Designing High-Bandwidth On-Chip Caches. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 121-132, June 1997.

[9] Standard Performance Council. The SPEC95 CPU Benchmark Suite. <http://www.spec.org/cpu2000>

[10] A.R. Alameldeen and D.A. Wood. Variability in Architectural Simulations of Multi-threaded Workload. In *Proceedings of the 9th Annual International Symposium on High Performance Computer Architecture*, pages 7-18, February 2003.

[11] J. Lo, L. A. Barroso, S. Eggers, K. Gharachorloo, H. Levy, and S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 39 - 50 June 1998.

[12] A. Ramirez, L. A. Barroso, K. A. Gharachorloo, R. Cohn, J. Larriba-Pey, P. G. Lowney, M. Valero. Code Layout Optimizations for Transaction Processing Workloads. In *Proceedings of the 28th Intl. Symposium on Computer Architecture*, pages 155-164, June 2001.

[13] P.S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner. SimICS/sun4m: A Virtual Workstation. In *Proceedings of the Usenix Annual Technical Conference*, pages 119-130, June 1998.