In Proceedings of ASPLOS-VIII (October 1998)

Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors

Parthasarathy Ranganathan*, Kourosh Gharachorloo†, Sarita V. Adve*, and Luiz André Barroso†

*Electrical and Computer Engineering
Rice University
{*parthas,sarita*}@*rice.edu*

†Western Research Laboratory
Compaq Computer Corporation
{*barroso,kourosh*}@*pa.dec.com*

Abstract

Database applications such as online transaction processing (OLTP) and decision support systems (DSS) constitute the largest and fastest-growing segment of the market for multiprocessor servers. However, most current system designs have been optimized to perform well on scientific and engineering workloads. Given the radically different behavior of database workloads (especially OLTP), it is important to re-evaluate key system design decisions in the context of this important class of applications.

This paper examines the behavior of database workloads on shared-memory multiprocessors with aggressive out-of-order processors, and considers simple optimizations that can provide further performance improvements. Our study is based on detailed simulations of the Oracle commercial database engine. The results show that the combination of out-of-order execution and multiple instruction issue is indeed effective in improving performance of database workloads, providing gains of 1.5 and 2.6 times over an in-order single-issue processor for OLTP and DSS, respectively. In addition, speculative techniques enable optimized implementations of memory consistency models that significantly improve the performance of stricter consistency models, bringing the performance to within 10-15% of the performance of more relaxed models.

The second part of our study focuses on the more challenging OLTP workload. We show that an instruction stream buffer is effective in reducing the remaining instruction stalls in OLTP, providing a 17% reduction in execution time (approaching a perfect instruction cache to within 15%). Furthermore, our characterization shows that a large fraction of the data communication misses in OLTP exhibit migratory behavior; our preliminary results show that software prefetch and writeback/flush hints can be used for this data to further reduce execution time by 12%.

1 Introduction

With the increasing demand for commercial applications, database workloads such as online transaction processing (OLTP) and decision support systems (DSS) have quickly surpassed scientific and engineering workloads to become the largest market segment for multiprocessor servers. While the behavior of DSS workloads has been shown to be somewhat reminiscent of scientific/engineering applications [2, 28], a number of recent studies have underscored the radically different behavior of OLTP workloads [2, 4, 5, 11, 14, 20, 21]. In general, OLTP workloads lead to inefficient executions with a large memory stall component and present a more challenging set of requirements for processor and memory system design. This behavior arises from large instruction and data footprints and high communication miss rates that are characteristic for such workloads [2].

The dramatic change in the target market for shared-memory servers has yet to be fully reflected in the design of these systems. Current processors have been primarily optimized to perform well on the SPEC benchmark suite [24], and system designs are focused on scientific and engineering benchmarks such as STREAMS [15] and SPLASH-2 [31]. One important outcome of this trend has been the emergence of aggressive out-of-order processors that exploit instruction-level parallelism (ILP) with ever-increasing design complexity. Given the dominant role of database workloads in the marketplace, it is important to re-evaluate the benefits of ILP features such as out-of-order execution, multiple instruction issue, non-blocking loads, and speculative execution in the context of such workloads. The goal of this paper is to shed light on the benefits of such techniques for database applications, thus helping designers determine whether the benefits warrant the extra system complexity.

This paper presents a detailed simulation study of database workloads running on shared-memory multiprocessors based on next-generation out-of-order processors. We present a thorough analysis of the benefits of techniques such as out-of-order execution and multiple issue in database applications, and identify simple solutions that further optimize the performance of the more challenging OLTP workload. In contrast, most previous studies of aggressive out-of-order processors in shared-memory systems have focused on scientific and engineering applications. Similarly, architectural studies of database workloads have been mostly based on simple in-order processor models [2, 5, 28].

To investigate the behavior of databases, we have instrumented and studied the Oracle commercial database engine (version 7.3.2) running on Alpha processors under Digital Unix. We use traces of OLTP and DSS workloads running on Oracle to drive a highly detailed trace-driven multiprocessor simulator. Our base set of results show that the combination of out-of-order execution and multiple issue provide performance improvements of 1.5 and 2.6 times for OLTP and DSS, respectively, over multiprocessor systems with single-issue in-order processors. While multiple issue and out-of-order execution individually improve performance, the combination of these techniques interact synergistically to achieve higher performance.

Given the range of memory consistency models supported by commercial multiprocessors (sequential consistency for SGI, "processor consistency"-like for Intel and Sun, and Alpha or PowerPC consistency for Digital and IBM), we were also interested in evaluating the effectiveness of speculative techniques that can be used in conjunction with out-of-order processors [7]. Our results show that these techniques can reduce the execution time of OLTP and DSS in sequentially consistent systems by 26-37%, bringing the performance to within 10-15% of systems with more relaxed models (e.g., Alpha consistency). Given that these techniques have been adopted in several commercial microprocessors (e.g., HP PA8000, Intel Pentium Pro, Mips R10000), the choice of the hardware consistency model for a system does not seem to be a dominant factor for database workloads, especially for OLTP.

The second part of our study focuses on further reducing the remaining instruction stall and remote memory latencies in our OLTP workload. We show that a simple 4-entry instruction stream buffer can reduce the execution time by 17%, bringing the performance to within 15% of a system with a perfect instruction cache. For data misses, our results indicate that the memory stall time is dominated by the latency of remote dirty misses. Our characterization shows that most of these data misses are generated by a small subset of the application instructions and exhibit migratory data access patterns. In the absence of source code for Oracle, we used a simple scheme for inserting software prefetch and flush/write-back hints into the code as a preliminary study. This optimization provided a 12% reduction in execution time.

The rest of paper is structured as follows. The next section presents our experimental methodology, including a description of OLTP and DSS workloads and the simulated multiprocessor system. Section 3 describes the base performance results with aggressive out-of-order processors. We address the remaining instruction- and data-related memory stalls for OLTP in Section 4. Finally, we discuss related work and conclude.

2 Experimental Methodology

Because of their complexity and size, commercial-grade database workloads are challenging applications to study in a simulation environment. In this section, we describe our database workloads and the experimental setup used to study them.

2.1 Database Workloads

We use the Oracle 7.3.2 commercial database management system as our database engine. In addition to the server processes that execute the actual database transactions, Oracle spawns a few daemon processes that perform a variety of duties in the execution of the database engine. Two of these daemons, the database writer and the log writer, participate directly in the execution of transactions. The database writer daemon periodically flushes modified database blocks that are cached in memory out to disk. The log writer daemon is responsible for writing transaction logs to disk before it allows a server to commit a transaction.

Client processes communicate with server processes through pipes, and the various Oracle processes (i.e., daemons and servers) communicate through a shared memory region called the System Global Area (SGA). The SGA consists of two main regions - the block buffer area and the metadata area. The block buffer area is used as a memory cache of database disk blocks. The metadata area is used to keep directory information for the block buffer, as well as for communication and synchronization between the various Oracle processes.

2.1.1 OLTP Workload

Our OLTP application is modeled after the TPC-B benchmark from the Transaction Processing Performance Council (TPC) [29]. TPC-B models a banking database system that keeps track of customers' account balances, as well as balances per branch and teller. Each transaction updates a randomly chosen account balance, which includes updating the balance of the branch the customer belongs to and the teller from which the transaction is submitted. It also adds an entry to the history table which keeps a record of all submitted transactions.

The application was extensively tuned in order to maximize transaction throughput and CPU utilization. For OLTP, we run Oracle in "dedicated mode," in which each client process has a dedicated Oracle server process to execute database transactions.

We chose to use TPC-B instead of TPC-C (the current official transaction processing benchmark from TPC) for a variety of reasons. First, TPC-B has much simpler setup requirements than TPC-C, and therefore lends itself better for experimentation through simulation. Second, our performance monitoring experiments with TPC-B and TPC-C show similar processor and memory system behavior, with TPC-B exhibiting somewhat worse memory system behavior than TPC-C [2]. As a result, we expect that changes in processor and memory system features to affect both benchmarks in similar ways. Finally, it is widely acknowledged that actual customer database applications will typically show poorer performance than TPC-C itself.

2.1.2 DSS Workload

The DSS application is modeled after Query 6 of the TPC-D benchmark [30]. The TPC-D benchmark represents the activities of a business that sells a large number of products on a worldwide scale. It consists of several inter-related tables that keep information such as parts and customer orders. Query 6 scans the largest table in the database to assess the increase in revenue that would have resulted if some discounts were eliminated. The behavior of this query is representative of other TPC-D queries [2].

For DSS, we used Oracle with the Parallel Query Optimization option, which allows the database engine to decompose the query into multiple sub-tasks and assign each one to an Oracle server process. The queries were parallelized to generate four server processes per processor (16 processes in a 4-processor system).

2.2 Simulation Methodology

We use the RSIM simulation infrastructure [17] to model multiprocessor systems with processors that exploit ILP techniques. Due to the difficulty of running a commercial-grade database engine on a user-level simulator (such as RSIM), our strategy was to use traces of the applications running on a four-processor AlphaServer4100, and drive the simulator with those traces. This trace-driven simulation methodology is similar to that used by Lo et al. [13].

The traces were derived with a custom tool built using ATOM [23]. Only the Oracle server processes were traced since the many daemon processes have negligible CPU requirements. However, the behavior of the daemons with respect to synchronization and I/O operations was preserved in the traces. All blocking system calls were marked in the traces and identified as hints to the simulator to perform a context switch. The simulator uses these hints to guide context switch decisions while internally modeling the operating system scheduler. The simulation includes the latency of all I/O and blocking system calls. The values for these latencies were determined by instrumenting the application to measure the effect of the system calls on an Alpha multiprocessor.

The trace also includes information regarding Oracle's higher-level synchronization behavior. The values of the memory locations used by locks are maintained in the simulated environment. This enables us to correctly model the synchronization between processes in the simulated environment and avoid simulating spurious synchronization loops from the trace-generation environment. Our results show that most of the lock accesses in OLTP were contentionless and that the work executed by each process was relatively independent of the order of acquisition of the locks. DSS shows negligible locking activity.

One trace file was generated per server process in the system. The total number of instructions simulated was approximately 200 million for both OLTP and DSS. Warmup transients were ignored in the statistics collection for both the workloads.

2.3 Scaling and Validation

We followed the recommendations of Barroso et al. [2] in scaling our workloads to enable tracing and simulation. Specifically, we carefully scaled down our database and block buffer sizes while continuing to use the same number of processes per processor as a full-sized database. We use an OLTP database with 40 branches and an SGA size over 900MB (the size of the metadata area is over 100MB). The DSS experiments use an in-memory 500MB database. The number of processes per CPU was eight for OLTP and four for DSS. Similar configurations were used by Lo et al. [13].

In the past, transaction processing applications were reported to be mainly I/O bound and to have a dominant component of their execution time in the operating system. Today, a modern database engine can tolerate I/O latencies and incurs much less operating system overhead; the operating system component for our tuned workloads (measured on the AlphaServer4100) was less than 20% of the total execution time for the OLTP workload and negligible for the DSS workload. Since our methodology uses user-level traces, we do not take into account the non-negligible operating system overheads of OLTP. However, as reported in Barroso et al. [2], the execution behavior of Digital Unix running this OLTP workload is very similar to the user-level behavior of the application, including CPI, cache miss ratios, and contributions of different types of misses. Therefore, we expect that the inclusion of operating system activity would not change the manner in which our OLTP workload is affected by most of the optimizations studied here.

Significant care was taken to ensure that the traces accurately reflect the application behavior, and that the simulated execution reproduces the correct interleaving of execution and synchronization behavior of the various processes. We configured our simulator to model a configuration similar to that of our server platform and verified that the cache behavior, locking characteristics, and speedup of the simulated system were similar to actual measurements of the application running on our server platform. We also verified our statistics with those reported in [2] and [13] for similar configurations.

2.4 Simulated Architecture

We use RSIM to simulate a hardware cache-coherent non-uniform memory access (CC-NUMA) shared-memory multiprocessor system using an invalidation-based, four-state MESI directory coherence protocol. Due to constraints of simulation time, we only model a system with four nodes. Each node in our simulated system includes a processor, separate first level data and instruction caches, a unified second-level cache, a portion of the global shared-memory and directory, and a network interface. A split-transaction bus connects the network interface, directory controller, and the rest of the system node. The system uses a two-dimensional wormhole-routed mesh network.

The L1 data cache is dual-ported, and uses a write-allocate, write-back policy. The unified L2 cache is a fully pipelined, write-allocate write-back cache. In addition, all caches are non-blocking and allow up to 8 outstanding requests to separate cache lines. At each cache, miss status holding registers (MSHRs) [12] store information about the misses and coalesce multiple requests to the same cache line. All caches are physically addressed and physically tagged. The virtual memory system uses a bin-hopping page mapping policy with 8K page sizes, and includes separate 128-element fully associative data and instruction TLBs.

Our base system models an out-of-order processor with support for multiple issue, out-of-order instruction execution, non-blocking loads, and speculative execution. We use an aggressive branch prediction scheme that consists of a hybrid *pa/g* branch predictor for the conditional branches [26], a branch target buffer for the jump target branches, and a return address stack for the call-return branches. In the event of branch mispredictions, we do not issue any instructions from after the branch until the branch condition is resolved; our trace-driven methodology precludes us from executing the actual instructions from the wrong-path.

Figure 1 summarizes the other important parameters used in our base processor model. To study the effect of the individual techniques as well as the relative importance of various performance bottlenecks, we vary many of these parameters in our experiments. Specifically, we study both in-order and out-of-order processor models, and the effect of instruction window size, issue width, number of outstanding misses, branch prediction, number of functional units, and cache size on the performance.

Both the in-order and out-of-order processor models support a straightforward implementation of the Alpha consistency model (hereafter referred to as release consistency [RC] for ease of notation), using the Alpha `MB` and `WMB` fence instructions to impose ordering at synchronization points. The out-of-order processor model also supports implementations of sequential consistency (SC) and processor consistency (PC), and optimized implementations for these consistency models. These are further described in Section 3.4.

3 Impact of Aggressive Processor Features on Database Workloads

Sections 3.1 and 3.2 evaluate the performance benefits and limitations of aggressive ILP techniques for OLTP and DSS workloads. Section 3.3 provides a comparison of multiprocessor results with those for uniprocessors. Finally, Section 3.4 examines the performance of optimized implementations of memory consistency models enabled by ILP features.

3.1 Performance Benefits from ILP Features

Figures 2 and 3 present our results for OLTP and DSS respectively. Part (a) of each figure compares multiprocessor systems with in-order and out-of-order processors with varying issue widths. Part (b) shows the impact of increasing the instruction window size for the out-of-order processor. Parts (c) through (g) show the impact of supporting multiple outstanding misses (discussed later). The bars in each graph represent the execution time normalized to that of the leftmost bar in the graph.[1] We further breakdown execution

[1] We factor out the idle time in all the results; the idle time is less than 10% in most cases.

Processor parameters	
Processor speed	1 GHz
Issue width	4 (default)
Instruction window size	64 (default)
Functional units	
- integer arithmetic	2
- floating point	2
- address generation	2
Branch prediction	
- conditional branches	*PA(4K,12,1)/g(12,12)*
- jmp branches	512-entry 4-way BTB
- call-returns	32-element RAS
Simultaneous speculated branches	8
Memory queue size	32

Contentionless memory latencies	
Memory type	*Latency (in processor cycles)*
Local read	100
Remote read	160-180
Cache-to-Cache read	280-310

Memory hierarchy	
Cache line size	64 bytes
Number of L1 MSHRs	8
L1 data cache size (on-chip)	128 KB
L1 data cache associativity	2-way
L1 data cache request ports	2
L1 data cache hit time	1 cycle
L1 instruction cache size (on-chip)	128 KB
L1 instruction cache associativity	2-way
L1 instruction cache request ports	2
L1 instruction cache hit time	1 cycle
L2 cache size (off-chip)	8M
L2 cache associativity	4-way
L2 request ports	1
L2 hit time (pipelined)	20 cycles
Number of L2 MSHRs	8
Data TLB	128 entries, full-assos
Instruction TLB	128 entries, full-assos

Figure 1: Default system parameters.

time into CPU (both busy and functional unit stalls), data (both read and write), synchronization, and instruction stall (including instruction cache and iTLB) components. Additionally, the bars on the right hand side in parts (b) and (c) show a magnification of the read stall time corresponding to the bars on the left hand side. The read stall time is divided into L1 hits plus miscellaneous stalls (explained below), L2 hits, local and remote memory accesses (serviced by memory), dirty misses (i.e., cache-to-cache transfers), and data TLB misses. The base results assume a release-consistent system, therefore there is little or no write latency. Section 3.4 discusses the performance of stricter consistency models.

With out-of-order processors, it is difficult to assign stall time to specific instructions since each instruction's execution may be overlapped with both preceding and following instructions. We use the following convention to account for stall cycles. At every cycle, we calculate the ratio of the instructions retired that cycle to the maximum retire rate and attribute this fraction of the cycle to the busy time. The remaining fraction is attributed as stall time to the first instruction that could not be retired that cycle. For memory instructions, there are extra stall cycles that may be attributed to the instruction in addition to the time spent in the execution stage: (i) time spent in the address generation stage that cannot be hidden due to data dependence or resource contention, and (ii) pipeline restart time after a branch misprediction, instruction cache miss, or exception when the memory instruction is at the head of the window. While these stalls are included for all memory categories, their impact is particularly visible in the "L1+misc" component because the base L1 latency is only one cycle. Similar conventions have been adopted by many previous studies (e.g., [18, 21]).

Overall, our results show that the OLTP workload is characterized by a significant L2 component due to its large instruction and data footprint. In addition, there is a significant memory component arising from frequent data communication misses. For OLTP, the local miss rates for the first-level instruction and data caches and the second level unified cache are 7.6%, 14.1% and 7.4% respectively. In contrast, the main footprint for the DSS workload fits in the large L1 caches (128K), and the memory component is much smaller relative to OLTP; DSS is more compute intensive and benefits from spatial locality on L2 misses. The local miss rates for DSS are 0.0% and 0.9% for the first-level instruction and data caches and 23.1% for the second level cache. These observations are consistent with those reported in previous studies [2, 13].

The results further indicate that support for multiple issue, out-of-order execution, and multiple outstanding loads provide significant benefits for OLTP and DSS, even though the benefits for OLTP are smaller in comparison. Most of the gains are achieved by a configuration with four-way issue, an instruction window of 32 to 64 entries, and a maximum of four outstanding cache misses (to unique cache lines). Interestingly, many current processors are in fact more aggressive than this. For example, the HP-PA 8000 supports a fifty-six entry instruction window and ten outstanding misses. The Alpha 21264 supports an eighty entry instruction window and eight outstanding misses.

3.1.1 OLTP Workload

Multiple Issue

Going from single- to eight-way issue, in-order processors provide a 12% improvement in execution time while out-of-order processors provide a 22% improvement (Figure 2(a)). The benefits from multiple issue stem primarily from a reduction in the CPU component. Out-of-order processors allow more efficient use of the increased issue width. However, the benefits for in-order and out-of-order processors level off at two-way and four-way issue (respectively).

Out-of-order Execution

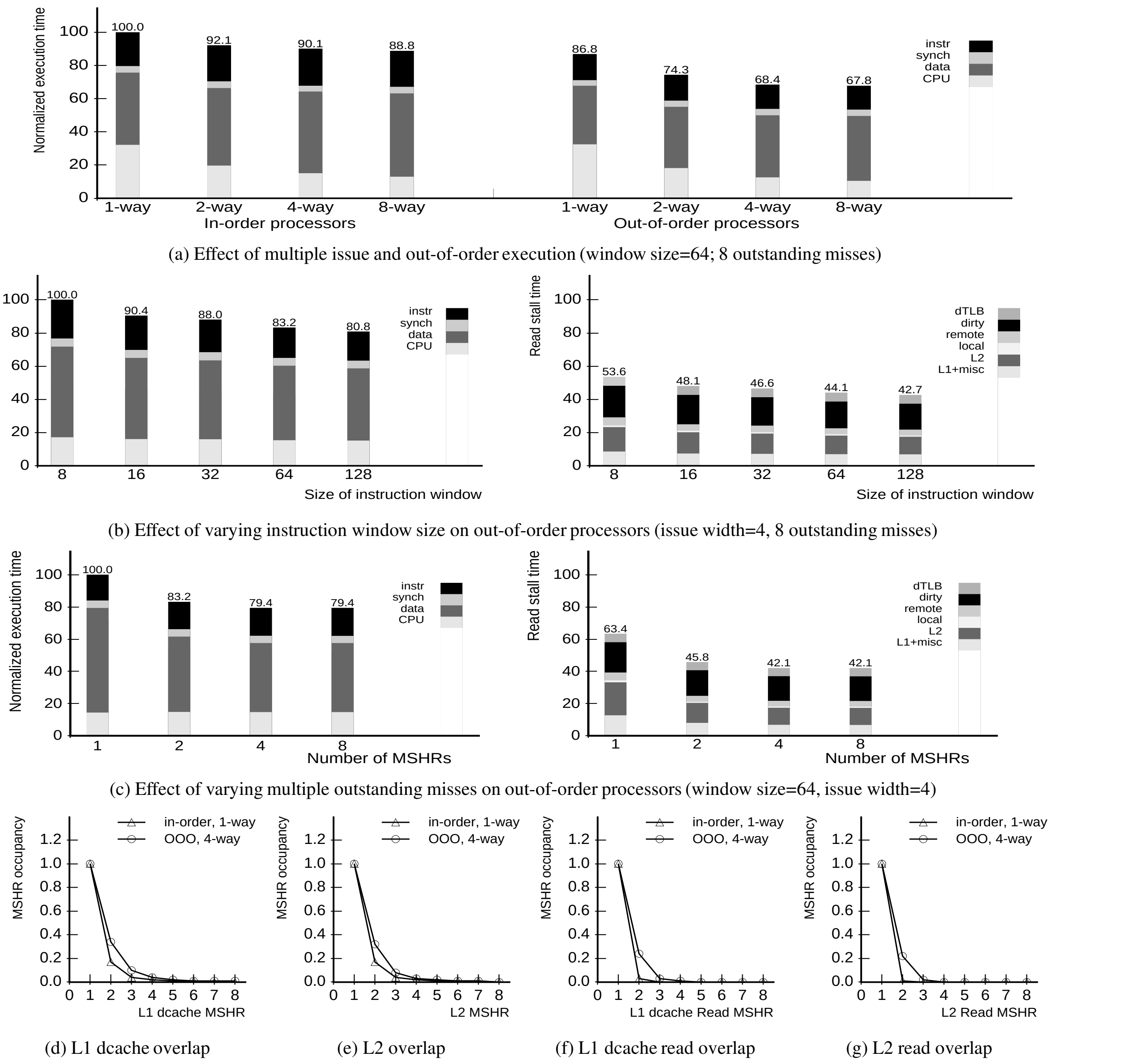
Comparing equivalent configurations with in-order and out-of-order processors in Figure 2(a), we see that out-of-order execution achieves reductions in execution time ranging from 13% to 24%, depending on the issue width. The combination of multiple issue and out-of-order execution interact synergistically to achieve higher performance than the sum of the benefits from the individual features.

The performance improvements due to out-of-order execution stem from reductions in the instruction and data stall components of execution time. The decoupling of fetch and execute stages in out-of-order processors enables part of the instruction miss latency to be overlapped with the execution of previous instructions in the instruction window. Similarly, the latency of a data load operation can be overlapped with other operations in the instruction window. In contrast, the amount of overlap in in-order processors is fundamentally limited since these systems require the processor to stall at the first data dependence that is detected.

Increasing the instruction window size increases the potential for overlap in out-of-order processors. As seen from Figure 2(b), performance improves as the instruction window size is increased, but levels off beyond 64 entries. A large fraction of this improvement comes from the L2 cache hit component (the read stall time graph of Figure 2(b)).

Multiple Outstanding Misses

Figure 2(c) summarizes the impact of increasing the number of



(a) Effect of multiple issue and out-of-order execution (window size=64; 8 outstanding misses)

(b) Effect of varying instruction window size on out-of-order processors (issue width=4, 8 outstanding misses)

(c) Effect of varying multiple outstanding misses on out-of-order processors (window size=64, issue width=4)

(d) L1 dcache overlap

(e) L2 overlap

(f) L1 dcache read overlap

(g) L2 read overlap

Figure 2: Impact of ILP features on OLTP performance.

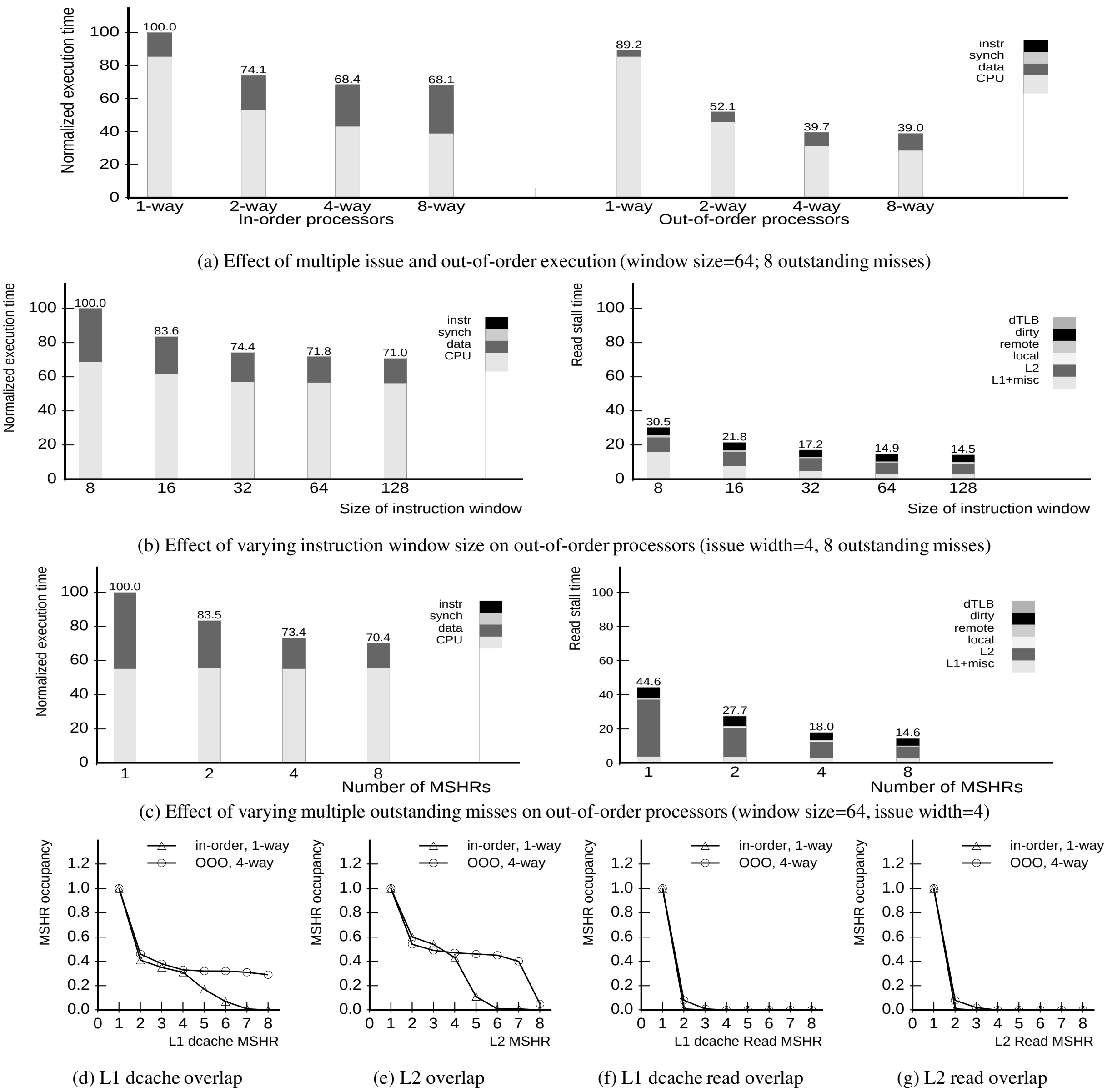
outstanding misses. The various bars show the performance as the number of MSHRs is increased. For OLTP, supporting only two outstanding misses achieves most of the benefits. This behavior is consistent with frequent load-to-load dependences that we have observed in OLTP.

To further understand this result, Figures 2(d)-(g) display MSHR occupancy distributions. The distribution is based on the total time when at least one miss is outstanding, and plots the fraction of this time that is spent when at least *n* MSHRs are in use. Figures 2(d) and (e) present the distributions for all misses at the first-level data and second-level unified caches, while Figures 2(f) and (g) correspond to only read misses.

Figures 2(f)-(g) indicate that there is not much overlap among read misses, suggesting that the performance is limited by the data-dependent nature of the computation. By comparing Figures 2(d) and (e) (both read and write misses) with Figures 2(f)-(g) (read misses), we observe that the primary need for multiple outstanding misses stems from writes that are overlapped with other accesses. Overall, we observe that there is a small increase in the MSHR occupancy when going from in-order to out-of-order processors which correlates with the decrease in the read stall times seen in Figure 2(c).

3.1.2 DSS Workload

Figure 3 summarizes the results for DSS. Overall, the improvements obtained from the ILP features are much higher for the DSS workload than for OLTP. This results from the compute-intensive nature of DSS that leads to little memory stall time. Clearly, the ILP features are more effective in addressing the non-memory stall times. Going from single issue to 8-way issue achieves a 32% reduction in execution time on in-order processors, and a 56% reduction in execution time on out-of-order processors (Figure 3(a)). Out-of-order issue achieves reductions in execution time ranging from 11% to 43% depending on the issue width. As with OLTP, there is synergy between out-of-order and multiple issue. Perfor-



(a) Effect of multiple issue and out-of-order execution (window size=64; 8 outstanding misses)

(b) Effect of varying instruction window size on out-of-order processors (issue width=4, 8 outstanding misses)

(c) Effect of varying multiple outstanding misses on out-of-order processors (window size=64, issue width=4)

(d) L1 dcache overlap

(e) L2 overlap

(f) L1 dcache read overlap

(g) L2 read overlap

Figure 3: Impact of ILP features on DSS performance.

mance levels off for instruction window sizes beyond 32. Figures 2(c) and 3(c) indicate that the DSS workload can exploit a larger number of outstanding misses (4 compared to 2 in OLTP). Figures 3(d)-(g) indicate that this is mainly due to write overlap. The high write overlap arises because of the relaxed memory consistency model we use which allows the processor to proceed past write misses without blocking.

3.2 Limitations of the ILP Features

3.2.1 OLTP Workload

Although the aggressive ILP features discussed above significantly improve OLTP performance, the execution time is still dominated by various stall components, the most important of which are instruction misses and dirty data misses. This leads to a low IPC of 0.5 on the base out-of-order processor.

We next try to determine if enhancement of any processor features can alleviate the remaining stall components. Our results are summarized in Figure 4. The left-most bar represents the base out-of-order configuration; subsequent bars show the effect of infinite functional units, perfect branch prediction, and a perfect instruction cache. The last bar represents a system with twice the instruction window size (128 elements) along with infinite functional units and perfect branch prediction, instruction cache, and i- and d-TLBs (Figure 2(b) shows the performance of the system when the instruction window size alone is doubled).

The results clearly show that functional units are not a bottleneck. Even though OLTP shows a cumulative branch misprediction rate of 11%, perfect branch prediction gives only an additional 6% reduction in execution time. Frequent instruction misses prevent the branch prediction strategy from having a larger impact. Not surprisingly, an infinite instruction cache gives the largest gain, illustrating the importance of addressing the instruction stall time. Increasing the instruction window size on a system with infinite functional units, perfect branch prediction, perfect instruction cache,

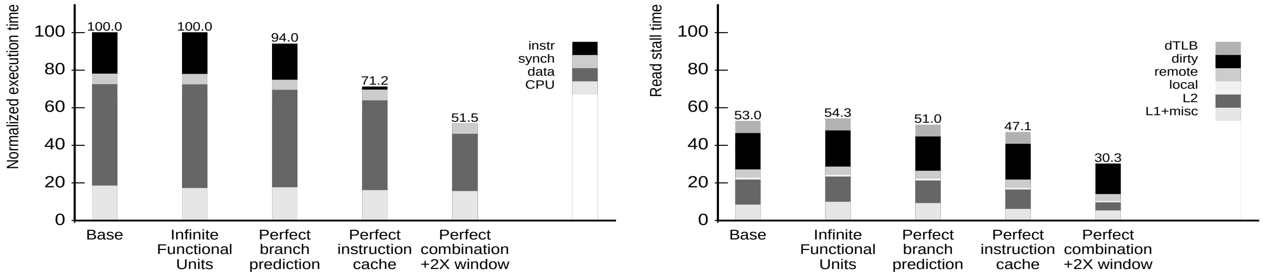
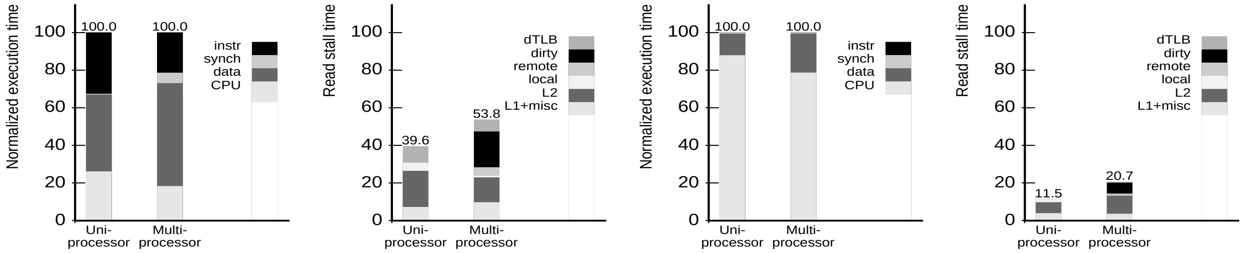


Figure 4: Factors limiting OLTP performance.



(a) OLTP workload

(b) DSS workload

Figure 5: Relative importance of components in uniprocessor and multiprocessor systems.

and perfect TLB behavior (rightmost bar in Figure 4) allows for more synergistic benefits. The L2 stall component is further diminished, leaving dirty miss latencies as the dominant component. Section 4 discusses techniques to address both the instruction stall and the dirty miss components.

3.2.2 DSS Workload

As discussed before, the DSS workload experiences very little stall time due to its highly compute-intensive nature and its relatively small primary working set. The IPC of the DSS workload on our base out-of-order system is 2.2. The main limitation in this application is the number of functional units (results not shown). Increasing the number of functional units (to 16 ALUs and 16 address generation units; floating point units are not used by the workload) results in a 12% performance improvement. For our default configuration, improving other parameters like the branch prediction, instruction cache and tlb sizes do not make any significant impact.

3.3 Comparison with Uniprocessor Systems

Our experiments show that uniprocessor systems achieve benefits quantitatively similar to multiprocessors from ILP features for both DSS and OLTP. However, it is interesting to compare the sources of stall time in uniprocessor and multiprocessor systems. Figure 5 presents the normalized execution times for our base uniprocessor and multiprocessor systems. For OLTP, the instruction stall time is a significantly larger component of execution time in uniprocessors since there are no data communication misses. For both workloads, multiprocessors bring about larger read components as expected.

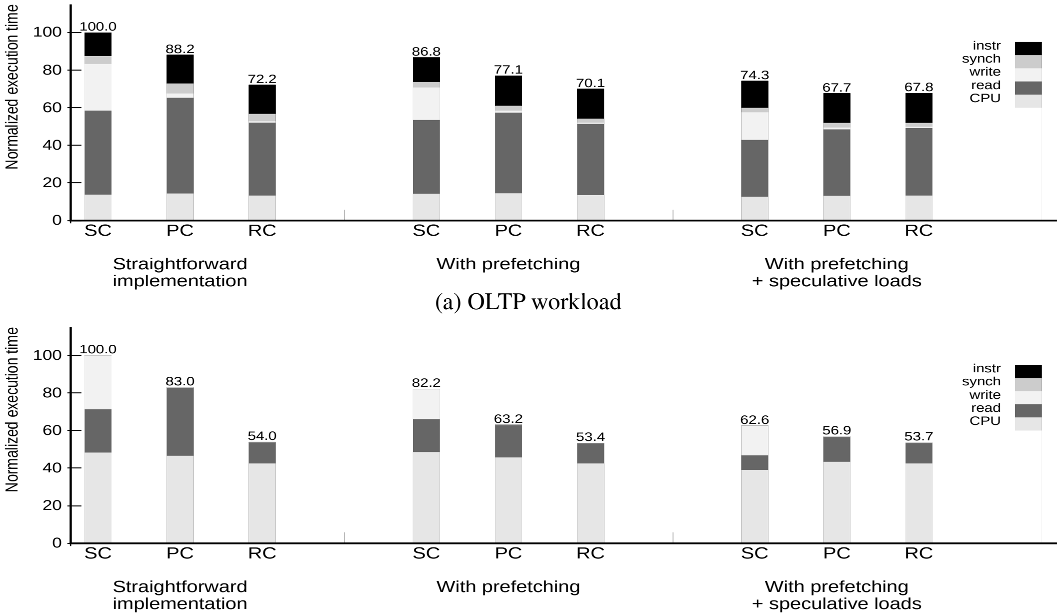
3.4 Performance Benefits from ILP-Enabled Consistency Optimizations

Features such as out-of-order scheduling and speculative execution also enable hardware optimizations that enhance the performance of memory consistency models. These optimized implementations exploit the observation that the system must only *appear* to execute memory operations according to the specified constraints of a model.

The technique of *hardware prefetching* from the instruction window [7] issues non-binding prefetches for memory operations whose addresses are known, and yet are blocked due to consistency constraints. *Speculative load execution* [7] increases the benefits of prefetching by allowing the return value of the load to be consumed early, regardless of consistency constraints. The latter technique requires hardware support for detecting violations of ordering requirements due to early consumption of values and for recovering from such violations. Violations are detected by monitoring coherence requests and cache replacements for the lines accessed by outstanding speculative loads. The recovery mechanism is similar to that used for branch mispredictions or exceptions. Both of the above techniques are implemented in a number of commercial microprocessors (e.g., HP PA8000, Intel Pentium Pro, and MIPS R10000).

Figure 6 summarizes the performance of three implementations of consistency models – a straightforward implementation, another with hardware load and store prefetching, and a third that also uses speculative loads. The figure shows the performance of sequential consistency (SC), processor consistency (PC), and release consistency (RC) for each of the implementations. Execution times are normalized to the straightforward implementation of SC. The data stall component of execution time is further divided into read and write stall times.

Our results show that the optimizations have a large impact on



(a) OLTP workload

(b) DSS workload

Figure 6: Performance benefits from ILP-enabled consistency optimizations.

the performance of the stricter models (SC and PC), and a relatively small impact on RC (as expected). While prefetching alone achieves some benefits, the data-dependent nature of computation allows for even greater benefits with speculative loads. For example, with both prefetching and speculative loads, the execution time of SC is reduced by 26% for OLTP and by 37% for DSS, bringing it to within 10% and 15% of the execution time of RC. In contrast, without these optimizations, RC achieves significant reductions in execution time compared to SC (46% for DSS and 28% for OLTP). Given that these optimizations are already included in several commercial microprocessors, our results indicate that the choice of the hardware consistency model may not be a key factor for out-of-order systems running database workloads (especially OLTP). In contrast, previous studies based on the same optimizations have shown a significant performance gap (greater than 15%) between SC and RC for scientific workloads [19].

3.5 Summary of ILP Benefits

Techniques such as multiple issue, out-of-order execution, and multiple outstanding loads provide significant benefits for both OLTP and DSS. The gains for DSS are more substantial as compared with OLTP. OLTP has a large memory system component that is difficult to hide due to the prevalence of dependent loads. Most of the benefits are achieved by a four-way issue processor with a window size of 32 to 64 and a maximum of four outstanding cache misses. Furthermore, ILP features present in current state-of-the-art processors allow optimized implementations of memory consistency models that significantly improve the performance of stricter consistency models (by 26-37%) for database workloads, bringing their performance to within 10-15% of more relaxed models.

4 Addressing Instruction and Data Bottlenecks in OLTP

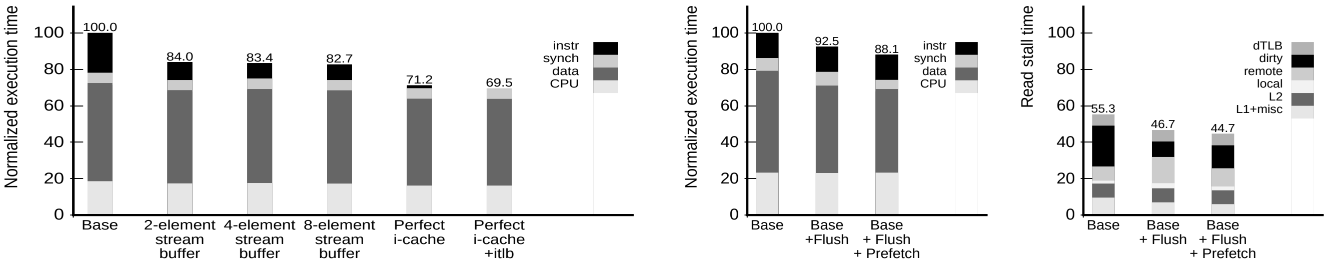
Section 3.2 showed that the instruction stall time and stall time due to read dirty misses are two primary bottlenecks that limit the performance of our OLTP workload. This section further analyzes these bottlenecks and evaluates simple solutions to address them.

4.1 Addressing the Instruction Bottleneck

Our analysis of the instruction stall time in OLTP identifies two key trends. First, the instruction stall time is dominated by first-level cache misses that hit in the second-level cache. This stems from the fact that the instruction working set of OLTP is about 560KB which fits in the large 8M second-level cache, but overwhelms the 128K first-level cache. Second, a significant portion of the instruction references follow a streaming pattern with successive references accessing successive locations in the address space. These characteristics suggest potential benefits from adding a simple instruction stream buffer between the first and second level caches.

A *stream buffer* is a simple hardware-based prefetching mechanism that automatically initiates prefetches to successive cache lines following a miss [10]. To avoid cache pollution due to the prefetched lines, the hardware stores the prefetched requests in the stream buffer until the cache uses the prefetched data. In the event of a cache miss that does not hit in any of the entries in the stream buffer, the hardware flushes all the entries in its buffer and initiates prefetches to the new stream.

Our results show that a simple stream buffer is very effective in reducing the effects of instruction misses in OLTP. A 2-element instruction buffer is successful in reducing the miss rate of the base out-of-order system by almost 64%. A 4-element stream buffer reduces the remaining misses by an additional 10%. Beyond 4 elements, the stream buffer provides diminishing returns for two rea-



(a) Addressing instruction misses

(b)Addressing communication misses
(base assumes 4-element stream buffer)

Figure 7: Addressing performance bottlenecks in OLTP.

sons. First, the misses targeted by the stream buffer decrease, since streams in OLTP are typically less than 4 cache lines long. Second, additional stream buffer entries can negatively impact performance by causing extra contention for second-level cache resources due to useless prefetches.

Figure 7(a) compares the performance of our base system to systems including stream buffers of size 2, 4, and 8 elements. As an upper bound on the performance achievable from this optimization, we also include results for a system with a perfect instruction cache, and a system with a perfect instruction cache and perfect instruction TLB.

As shown in Figure 7(a), a 2- or 4-element stream buffer reduces the execution time by 16%, bringing the performance of the system to within 15% of the configuration with a perfect icache. Most of the benefits come from increased overlap of multiple instruction misses at the L1 cache. The improvement in performance from stream buffers is more pronounced in uniprocessor configurations where the instruction stall time constitutes a larger component of the total execution time. Our results for uniprocessors (not shown here) show that stream buffers of size 2 and 4 achieve reductions in execution time of 22% and 27% respectively compared to the base system.

Further characterization of the instruction misses indicate that a large fraction of the remaining misses exhibit repeating sequences, though with no regular strides. Code realignment by the compiler, or a predictor that interfaces with a branch target buffer to issue prefetches for the right path of the branch could potentially target these misses. Our preliminary evaluation of the latter scheme indicates that the benefits from such predictors are likely to be limited by the accuracy of the path prediction logic and may not justify the associated hardware costs, especially when a stream buffer is already used.

An alternative to using a stream buffer is to increase the size of the transfer unit between the L1 and L2 caches. Our experiments with a 128-byte cache line size suggest that such an architectural change can also achieve reductions in miss rates comparable to the stream buffers. However, stream buffers have the potential to dynamically adapt to longer stream lengths without associated increases in the access latency, and without displacing other useful data from the cache.

We are not aware of any current system designs that include support for an instruction stream buffer between the L1 and L2 cache levels. Our results suggest that adding such a stream buffer can provide high performance benefits for OLTP workloads with marginal hardware costs.

4.2 Addressing the Data Communication Miss Bottleneck

As shown in Figure 5(a), read dirty misses (serviced by cache-to-cache transfers) account for 20% of the total execution time of OLTP on our base out-of-order system. In addition, dirty misses account for almost 50% of the total misses from the L2 cache. To better understand the behavior of these dirty misses, we performed a detailed analysis of the sharing patterns in Oracle when running our OLTP workload. Our key observations are summarized below.

First, we observed that 88% of all shared write accesses and 79% of read dirty misses are to data that exhibit a migratory sharing pattern.[2] OLTP is characterized by fine-grain updates of meta-data and frequent synchronization that protects such data. As a result, data structures associated with the most actively used synchronization tend to migrate with the passing of the locks.

Second, additional characterization of the migratory misses show that they are dominated by accesses to a small subset of the total migratory data, and are generated by a small subset of the total instructions. On our base system, 70% of the migratory write misses refer to 3% of all cache lines exhibiting migratory behavior (about 520 cache lines), and more importantly, 75% of the total migratory references are generated by less than 10% of all instructions that ever generate a migratory reference (about 100 unique instructions in the code). Finally, analysis of the dynamic occurrence of these instructions indicate that 74% of the migratory write accesses and 54% of the migratory read accesses occur within critical sections bounded by identifiable synchronization.

The above observations suggest two possible solutions for reducing the performance loss due to migratory dirty read misses. First, a software solution that identifies accesses to migratory data structures can schedule *prefetches* to the data, enabling the latency to be overlapped with other useful work. Support for such software-directed prefetch instructions already exists in most current processors.

Second, a solution that identifies the end of a sequence of accesses to migratory data can schedule a *"flush"* or *"WriteThrough"* [1] instruction to update the memory with the latest values. Subsequent read requests can then be serviced at memory, avoiding the extra latency associated with a cache-to-cache transfer (for our system configuration, this can reduce the latency by almost 40%). We found that it is important for the flush opera-

[2] We use the following heuristic to identify migratory data [3, 25]. A cache line is marked as migratory when the directory receives a request for exclusive ownership to a line, the number of cached copies of the line is 2, and the last writer to the line is not the requester. Because our base system uses a relaxed memory consistency model, optimizations for dealing with migratory data such as those suggested by Stenstrom et al. [25] will not provide any gains since the write latency is already hidden.

tion to keep a clean copy in the cache (i.e., not invalidate the copy) since the latency of subsequent read misses would otherwise neutralize the gains. This requires minor support in the protocol since the flush effectively generates an unsolicited "sharing writeback" message to the directory.

Given our lack of access to the Oracle source code, we could not correlate the migratory accesses to the actual data structures. Instead, we used our characterizations to identify the instructions that are likely to generate migratory data accesses to perform a preliminary study. Our experiments involve adding the appropriate prefetch and flush software primitives to the code around some of the key instructions.

Figure 7(b) summarizes our results. All results assume a 4-element instruction stream buffer given the importance of this optimization. The leftmost bar represents the base system with a 4-element stream buffer. The second bar represents adding appropriate flush primitives at the end of specific critical sections. As shown in Figure 7(b), the flush primitive is successful in significantly reducing the impact of dirty misses on the total execution time[3], achieving a 7.5% reduction in execution time. As an approximate bound on this improvement, we selectively reduced the latency of all migratory read accesses (by 40%) to reflect service by memory; this gave a bound of 9% which is very close to the 7.5% improvement from the flush primitive. Finally, also adding prefetching for migratory data at the beginning of critical sections (rightmost bar) provides a cumulative reduction of 12% in execution time. Late prefetches and contention effects due to the combination of flush and prefetch appear to limit additional performance benefits. A realistic software solution that is aware of the data structures which cause the migratory accesses may be able to avoid both these problems.[4]

5 Discussion and Related Work

Our work extends previous studies by providing a thorough analysis of the benefits of aggressive techniques such as multiple issue, out-of-order execution, non-blocking loads, and speculative execution in the context of database workloads, and by identifying simple optimizations that can provide a substantial improvement in performance. Our work is also the first to study the performance of memory consistency models in the context of database workloads.

There are a number of studies based on the performance of out-of-order processors for non-database workloads(e.g., [8, 16, 18]). Most previous studies of databases are based on in-order processors [2, 4, 5, 6, 14, 20, 27, 28], and therefore do not address the benefits of more aggressive processor architectures. A number of the studies are limited to uniprocessor systems [4, 6, 13, 14]. As discussed in Section 3, data communication misses play a more dominant role in multiprocessor executions and somewhat reduce the relative effect of instruction stall times.

Another important distinction among the database studies is whether they are based on monitoring existing systems [4, 5, 11, 27] (typically through performance counters) or based on simulations [6, 13, 14, 20, 21, 28]; one study uses a combination of both techniques [2]. Monitoring studies have the advantage of using larger scale versions of the workloads and allowing for a larger number of experiments to be run. However, monitoring studies are often limited by the configuration parameters of the system and the types of events that can be measured by the monitoring tools. For our own study, we found the flexibility of simulations invaluable for considering numerous design points and isolating the effects of different design decisions. Nevertheless, we found monitoring extremely important for tuning, scaling, and tracing our workloads and for verifying the results of our simulations.

The following describes the database studies based on out-of-order processors in more detail. Keeton et al. [11] present a monitoring study of the behavior of an OLTP workload (modeled after TPC-C) on top of Informix, using the performance counters on a quad Pentium Pro system. Similar to our study, this paper shows that out-of-order execution can achieve performance benefits on OLTP. Our work differs because we study a more futuristic processor configuration through detailed simulations. This enables us to quantitatively isolate the performance benefits from various ILP techniques as well as evaluate solutions to the various performance bottlenecks. We also evaluate the performance of DSS in addition to OLTP.

Rosenblum et al. [21] present a simulation study that focuses on the impact of architectural trends on operating system performance using three workloads, one of which is TPC-B on Sybase. The study considers both uniprocessor and multiprocessor systems, with the multiprocessor study limited to in-order processors. The TPC-B workload used in this study is not representative because it exhibits large kernel and idle components primarily due to the lack of sufficient server processes. The paper does however make the observation that the decoupling of the fetch and execute units allows for some overlap of the instruction stall time in the uniprocessor system. Lo et al. [13] examine the performance of OLTP and DSS on a simultaneous multithreaded (SMT) uniprocessor system with support for multiple hardware contexts using the same simulation methodology as our study. Even though the SMT functionality is added on top of an aggressive out-of-order processor, the study primarily focuses on the effects of SMT as opposed to the underlying mechanisms in the base processor.

We are not aware of any work other than the original stream buffer work by Jouppi [10] that has studied the effect of stream buffers to alleviate the instruction miss bottleneck. This may partly be due to the fact that the standard SPEC, SPLASH, and STREAM benchmark suites do not stress the instruction cache significantly. Our evaluation differs from Jouppi's [10] in two key respects. First, we evaluate the impact of the stream buffer optimization on the execution time of a complex commercially-used database engine. Second, we perform our evaluation in the context of an aggressive out-of-order processor model that already includes support to hide part of the instruction stall latencies and show that the stream buffer can still provide a benefit.

Primitives similar to the "flush" primitive that we use in Section 4 have been proposed and studied by a number of other groups (e.g., [9, 22]). Our "flush" primitive is modeled after the "WriteThrough" primitive used by Abdel-Shafi et al. [1]. That study also showed that the combination of prefetching and "WriteThrough" could be used to achieve better performance improvements than using either of them alone (in the context of scientific applications).

Our study is based on systems with conventional cache hierarchies. A number of processors from Hewlett-Packard have opted for extremely large off-chip first level instruction and data caches (e.g., HP PA-8200 with up to 2MByte separate first level instruction and data caches), which may be targeting the large footprints in database workloads. These very large first level caches make the use of out-of-order execution techniques critical for tolerating the correspondingly longer cache access times.

Finally, our study evaluates the benefits of exploiting intra-thread parallelism in database workloads. Two previous studies have focused on exploiting inter-thread parallelism through the use

[3] The increase in the local and remote components of read latency corresponds to the dirty misses that are converted to misses serviced by the memory.

[4] Given the amount of tuning that is done on database benchmarks, it would not be unthinkable for database vendors to include such optimizations in their code if it leads to significant gains.

of multithreaded processors [6, 13]. This approach depends on the fact that database workloads are already inherently parallel (either in the form of threads or processes) for hiding I/O latency. These studies show that multithreading can provide substantial gains, with simultaneous multithreading (SMT) [13] providing higher gains (as high as three times improvement for OLTP). Our study shows that DSS can benefit significantly from intra-thread parallelism (2.6 times improvement). The incremental gains from the addition of SMT are less significant in comparison [13]. Intra-thread parallelism is not as beneficial for OLTP (1.5 times improvement) due to the data dependent nature of the workload. In this case, SMT is more effective in hiding the high memory overheads.

6 Concluding Remarks

With the growing dominance of commercial workloads in the multiprocessor server market, it is important to re-evaluate the justifications for key design decisions that have been made primarily based on the requirements of scientific and engineering workloads. This paper provides an in-depth analysis of the performance of aggressive out-of-order processors in multiprocessor configurations, and considers simple optimizations that can provide further performance improvements, in the context of OLTP and DSS workloads.

Among database applications, online transaction processing (OLTP) workloads present the more demanding set of requirements for system designers. While DSS workloads are somewhat reminiscent of scientific/engineering applications in their behavior, OLTP workloads exhibit dramatically different behavior due to large instruction footprints and frequent data communication misses which often lead to cache-to-cache transfers. Our results reflect these characteristics, with our DSS workload achieving a 2.6 times improvement from out-of-order execution and multiple issue, and our OLTP workload achieving a more modest 1.5 times improvement. In addition to the frequent instruction and data misses, gains in OLTP are limited due to the data dependent nature of the computation.

The inefficiencies in OLTP may be addressed through a number of simple optimizations. We showed that a simple 4-entry stream buffer for instructions can provide an extra 17% reduction in execution time. Our preliminary results with software prefetch and flush hints are also promising, giving another 12% reduction in execution time.

We also found that speculative techniques that may be easily added to aggressive out-of-order processors are extremely beneficial for multiprocessor systems that support a stricter memory consistency model. For example, the execution time of a sequentially consistent system can be improved by 26% and 37% for OLTP and DSS workloads, bringing it to within 10-15% of more relaxed systems. Given that these techniques have been adopted in several commercial microprocessors, the choice of the hardware consistency model for a system does not seem to be a dominant factor for database workloads, especially for OLTP.

In the future, we are interested in further pursuing software optimizations (including prefetch and flush hints) especially in the context of OLTP workloads to see whether it is possible to further relieve some of the challenging requirements such workloads impose on hardware (e.g., large, fast off-chip caches and fast cache-to-cache transfers).

7 Acknowledgements

This paper benefited from discussions with Norm Jouppi, Jack Lo, and Dan Scales, and from comments by the anonymous reviewers. We would also like to thank Jef Kennedy from Oracle for reviewing this manuscript, Marco Annaratone from WRL for supporting this work, and Drew Kramer from WRL for technical support. This research was primarily done while Parthasarathy Ranganathan was a summer intern at WRL. At Rice, he is also supported by a Lodieska Stockbridge Vaughan Fellowship. Sarita Adve's research is supported in part by an Alfred P. Sloan Research Fellowship, IBM, the National Science Foundation under Grant No. CCR-9410457, CCR-9502500, CDA-9502791, and CDA-9617383, and the Texas Advanced Technology Program under Grant No. 003604-025.

References

[1] H. Abdel-Shafi, J. Hall, S. V. Adve, and V. S. Adve. An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pages 204–215, February 1997.

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

[3] A. L. Cox and R. J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.

[4] Z. Cventanovic and D. Bhandarkar. Performance characterization of the Alpha 21164 microprocessor using TP and SPECworkloads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 60–70, Apr 1994.

[5] Z. Cvetanovic and D. D. Donaldson. AlphaServer 4100 performance characterization. *Digital Technical Journal*, 8(4):3–20, 1996.

[6] R. J. Eickemeyer, R. E. Johnson, S. R. Kunkel, M. S. Squillante, and S. Liu. Evaluation of multithreaded uniprocessors for commercial application environments. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 203–212, June 1996.

[7] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages I:355–364, August 1991.

[8] K. Gharachorloo, A. Gupta, and J. Hennessy. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 22–33, May 1992.

[9] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative shared memory: Software and hardware support for scalable multiprocessors. *TOCS*, 11(4):300–318, November 1993.

[10] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings 17th Annual International Symposium on Computer Architecture*, pages 364–373, June 1990.

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

[12] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–85, 1981.

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

[14] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, Oct 1994.

[15] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. In *IEEE Technical Committee on Computer Architecture Newsletter*, Dec 1995.

[16] B. A. Nayfeh, L. Hammond, and K. Olukotun. Evaluation of Design Alternatives for a Multiprocessor Micro processor. In *Proceedings of the 23rd International Symp. on Computer Architec ture*, pages 67–77, May 1996.

[17] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM Reference Manual version 1.0. Technical Report 9705, Department of Electrical and Computer e University, August 1997.

[18] V. S. Pai, P. Ranganathan, and S. V. Adve. The Impact of Instruction Level Parallelism on Multiprocessor Performance and Simulation Methodology. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, pages 72–83, February 1997.

[19] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, Oct. 1996.

[20] S. E. Perl and R. L. Sites. Studies of windows NT performance using dynamic execution traces. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, pages 169–184, Oct. 1996.

[21] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 285–298, 1995.

[22] J. Skeppstedt and P. Stenstrom. A Compiler Algorithm that Reduces Read Latency in Ownership-Based Cache Coherence Protocols. In *International Conference on Parallel Architectures and Compilation Techniques*, 1995.

[23] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analys is Tools. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Languages*, March 1994.

[24] Standard Performance Council. *The SPEC95 CPU Benchmark Suite*. http://www.specbench.org, 1995.

[25] P. Stenstrom, M. Brorsson, and L. Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[26] T.-Y.Yeh and Y.N.Patt. Alternative Implementations of Two-level Adaptive Branch Prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[27] S. S. Thakkar and M. Sweiger. Performance of an OLTP application on Symmetry multiprocessor system. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 228–238, June 1990.

[28] P. Trancoso, J.-L. Larriba-Pey, Z. Zhang, and J. Torrellas. The memory performance of DSS commercial workloads in shared-memory multiprocessors. In *Third International Symposium on High-Performance Computer Architecture*, Jan 1997.

[29] Transaction Processing Performance Council. *TPC Benchmark B (Online Transaction Processing) Standard Specification*, 1990.

[30] Transaction Processing Performance Council. *TPC Benchmark D (Decision Support) Standard Specification*, Dec 1995.

[31] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.