STEPS Towards Cache-Resident Transaction Processing

Stavros Harizopoulos
Carnegie Mellon University
stavros@cs.cmu.edu

Anastassia Ailamaki
Carnegie Mellon University
natassa@cs.cmu.edu

Abstract

Online transaction processing (OLTP) is a multi-billion dollar industry with high-end database servers employing state-of-the-art processors to maximize performance. Unfortunately, recent studies show that CPUs are far from realizing their maximum intended throughput because of delays in the processor caches. When running OLTP, instruction-related delays in the memory subsystem account for 25 to 40% of the total execution time. In contrast to data, instruction misses cannot be overlapped with out-of-order execution, and instruction caches cannot grow as the slower access time directly affects the processor speed. The challenge is to alleviate the instruction-related delays without increasing the cache size.

We propose *Steps*, a technique that minimizes instruction cache misses in OLTP workloads by multiplexing concurrent transactions and exploiting common code paths. One transaction paves the cache with instructions, while close followers enjoy a nearly miss-free execution. *Steps* yields up to 96.7% reduction in instruction cache misses for each additional concurrent transaction, and at the same time eliminates up to 64% of mispredicted branches by loading a repeating execution pattern into the CPU. This paper (a) describes the design and implementation of *Steps*, (b) analyzes *Steps* using microbenchmarks, and (c) shows *Steps* performance when running TPC-C on top of the Shore storage manager.

1 Prologue

In the past decade, research has proposed techniques to identify and reduce CPU performance bottlenecks in database workloads. As memory access times improve much slower than processor speed, performance is bound by instruction and data cache misses that cause expensive main-memory accesses. Research [AD+99][LB+98][SBG02] shows that decision-support (DSS) applications are predominantly delayed by *data cache misses*, whereas OLTP is bounded by *instruction cache misses.* Although several techniques can reduce data cache misses (larger caches, out-of-order execution, better data placement), none of these can effectively address instruction caches.

1.1 Instruction cache behavior in OLTP

To maximize first-level instruction cache (L1-I cache) utilization and minimize stalls, application code should have few branches (exhibiting high *spatial locality*), a repeating pattern when deciding whether to follow a branch (yielding low *branch misprediction rate)*, and most importantly, the "working set" code footprint should fit in the L1-I cache. Unfortunately, OLTP workloads exhibit the exact opposite behavior [KP+98]. A study on Oracle reports a 556KB OLTP code footprint [LB+98]. With modern CPUs having 16-64KB L1-I cache sizes, OLTP code paths are too long to achieve cache-residency. Moreover, the importance of L1-I cache stalls increases with larger L2 caches (Fig. 1a, stalls shown as non-overlapping components; I-cache stalls are actually 41% of the total execution time [KP+98]). As a large L1-I cache may adversely impact the CPU's clock frequency, chip designers cannot increase L1-I sizes despite the growth in secondary caches (Fig. 1b). The increasing gap between cache levels makes L1-I cache misses the most important CPU stall factor in OLTP.

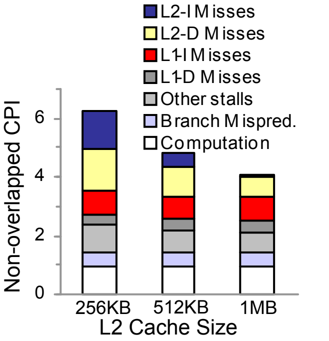


Figure 1a. *TPC-C CPU stall breakdown on PentiumPro. With larger L2 cache size L1-I misses become the dominant stall factor (3rd box from top).* [KP+98]

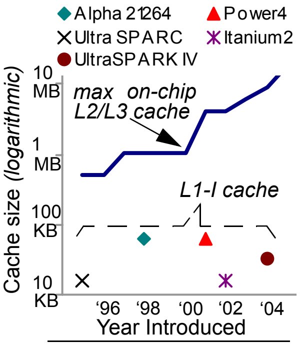
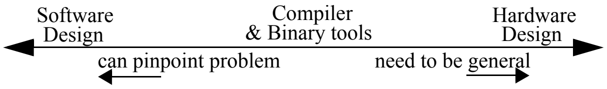


Figure 1b. *A decade-spanning trend shows that L1-I caches do not grow, while secondary, on-chip caches become increasingly larger.*

**Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004**

1.2 Related research

The full spectrum of approaches to improve instruction cache performance includes the following three areas:



At the hardware end, chip designers study database workload behavior and respond with higher-performance caches, but they are bound by design restrictions on the L1-I cache size [HP96]. Hardware enhancements apply to all workloads, and thus cannot effectively target a specific weakness of database workloads. At the binary representation level, compilers optimize DBMS code for a given set of hardware architectures [RV+97]. This area studies instruction traces and focuses on increasing the cache hit rate by reorganizing the binary code [RB+01]. Compilers still cannot "see" the root of the problem and thus, can partially alleviate L1-I cache misses only for statically trained workload instances.

A software designer has the best insight as to why the program incurs cache misses. For example, studying data cache access patterns and changing the memory page layout proved to be a key factor for reducing data cache misses [AD+01]. While the payoff in software approaches may be larger, improving instruction cache behavior for the entire code (typically millions of lines) is a great challenge. To our knowledge, this paper is the first to address instruction cache misses in transaction processing by proposing small changes in the DBMS code.

1.3 *Steps* to cache-resident code

We propose to exploit the high degree of concurrency that characterizes transaction processing workloads to maximize instruction sharing in the cache across different transactions. Consider, for instance, a hundred concurrent transactions executing the same high-level operation, e.g., record lookup using a B-tree index. Since the code working set typically overwhelms the L1-I cache and context switching occurs at random points, each transaction execution incurs new instruction misses.

To alleviate the problem, we propose a technique called *Synchronized Transactions through Explicit Processor Scheduling*, or *Steps*. *Steps* allows *only one* transaction to incur the *compulsory* instruction misses, while the rest of the transactions can piggyback in order to always find the instructions they need in the cache. To achieve this, *Steps* identifies the points in the code the cache fills up and performs a quick context-switch so that other transactions can execute the cache-resident code. We implemented *Steps* inside the Shore storage manager [Ca+94]. Figure 2a shows that, when running a group of transactions performing an indexed selection on a real machine with 64KB instruction cache, L1-I cache misses are reduced by 96.7% for each additional concurrent thread.

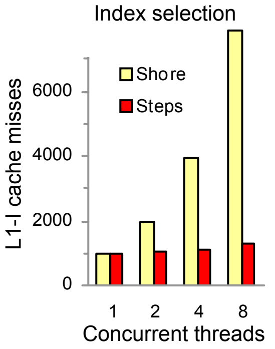


Figure 2a. *For a group of threads performing a tuple retrieval on a similar index,* Steps *practically eliminates additional L1-I misses.*

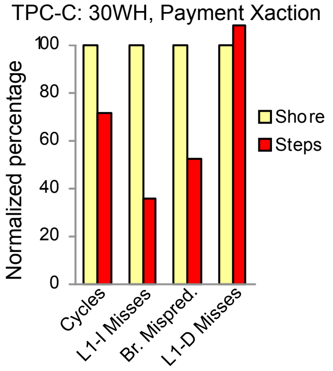


Figure 2b. *When 300 users run the TPC-C* Payment *transaction on a dataset of 30 Warehouses,* Steps *outperforms Shore on all of the time-critical events.*

A real transactional workload, however, includes different types of transactions and indices, involves mechanisms for logging and deadlock detection, and its execution is unpredictable due to frequent I/O and lock requests. *Steps* is designed to work with any OLTP workload; we illustrate our results using the widely accepted transactional benchmark TPC-C [Gra93]. Figure 2b shows that *Steps* running the Payment transaction in a 30-Warehouse configuration reduces instruction misses by 65% and mispredicted branches by 48%, while at the same time produces a 39% speedup on a real machine with 64KB L1-I cache. *Steps* incurs a 8% increase in the number of first-level data cache (L1-D) misses, which does not affect performance because the total number of L1-D cache misses is low and the penalty is easily hidden by instruction-level parallelism and out-of-order execution [AD+99].

1.4 Contributions and paper organization

This paper proposes a software technique to minimize instruction-related stalls for transactional database workloads. We demonstrate the validity and usefulness of *Steps* using the TPC-C benchmark running on Shore, a prototype state-of-the-art storage manager with similar behavior to commercial database systems [AD+01]. We use both real hardware (two different processors) and a full-system simulator. We focus on performance metrics that are not affected by the specifics of our system, and most importantly, we report *aggregate* (full-system) results for the execution time and cache usage statistics[1].

The paper is organized as follows. Section 2 provides background on processor caches and related work. Section 3 explains the implementation of *Steps* and demonstrates the benefits using microbenchmarks. Section 4 graduates *Steps* to the real world, removing all assumptions and running a full-fledged OLTP benchmark (TPC-C). We also describe how to apply *Steps* on any DBMS architecture.

[1] Full-system evaluation is crucial: results on isolated algorithms rarely reflect equal benefits when run inside a DBMS.

2 Background and related work

To bridge the CPU/memory performance gap, today's processors employ a hierarchy of caches that maintain recently referenced instructions and data close to the processor. Figure 3 shows an example of an instruction cache organization and explains the difference between *capacity* and *conflict* cache misses. Recent processors — e.g., IBM's Power4 — have up to three cache levels. At each hierarchy level, the corresponding cache trades off lookup speed for size. For example, level-one (L1) caches at the highest level are small (e.g., 16KB-64KB), but operate at processor speed. In contrast, lookup in level-two (L2) caches typically incurs up to an order of magnitude longer time because they are several times larger than the L1 caches (e.g., 512K-8MB). L2 lookup, however, is still several orders of magnitude faster than memory accesses (typically 300-400 cycles). Therefore, the effectiveness of cache hierarchy is extremely important for performance.

In contrast to data cache, instruction cache accesses are serialized and cannot be overlapped. Instruction cache misses prevent the flow of instructions through the processor and directly affect performance. Current trends towards improving process performance are leading to (i) increased on-chip L2 cache sizes (Figure 1b), and to (ii) increased degree of instruction-level parallelism through increasingly wider superscalar pipelines. In future processors, the combined effect of these two trends will result in significantly reduced cache data stalls (because multiple data cache accesses can be overlapped in parallel) making instruction cache stalls the key performance bottleneck.

2.1 Database workloads on modern processors

Prior research [MDO94] indicates that adverse memory access patterns in database workloads result in poor cache locality and overall performance. Recent studies of OLTP workloads and DBMS performance on modern processors [AD+99][KP+98] narrow the primary memory-related bottlenecks to L1 instruction and L2 data cache misses. More specifically, Keeton et al measure an instruction-related stall component of 41% of the total execution time for Informix running TPC-C on a PentiumPro [KP+98]. When running transactional (TPC-B and TPC-C) and decision-support (TPC-H) benchmarks on top of Oracle on Alpha processors, instruction stalls account for 45% and 30% of the execution time, respectively [BGB98][SBG02]. A recent study of DB2 7.2 running TPC-C on Pentium III [SA04] attributes 22% of the execution time to instruction stalls.

Unfortunately, unlike DSS workloads, transaction processing involves a large code footprint and exhibits irregular data access patterns due to the long and complex code paths of transaction execution. In addition, concurrent request reduces the effectiveness of single-query optimizations [JK99]. Finally, OLTP instruction streams have strong data dependencies that limit instruction-level parallelism opportunity, and irregular program control flow that undermines built-in pipeline branch prediction mechanisms and increases instruction stall time.

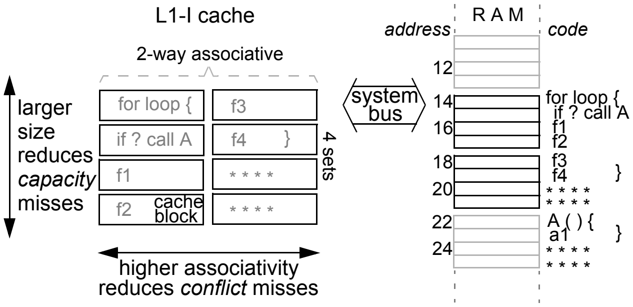


Figure 3. *Example of a 2-way set associative, 4-set (8 cache blocks) L1-I cache. Code stored in RAM maps to one set of cache blocks and is stored to any of the two blocks in that set. For simplicity we omit L2/L3 caches. In this example, the* for-loop *code fits in the cache only if procedure* A *is never called. In that case, repeated executions of the code will always hit in the L1-I cache. Larger code (more than eight blocks) would result in* capacity *misses. On the other hand, frequent calls to* A *would result to* conflict *misses because A's code would replace code lines f3 and f4 needed in the next iteration.*

2.2 Techniques to address L1-I cache stalls

In the last decade, research on *cache-conscious* database systems has primarily addressed data cache performance [SKN94][CGM01][GL01][AD+01]. L1-I cache misses, however, and misses occurring when concurrent threads replace each other's working sets [RB+95], have received little attention by the database community. A recent study [PM+01][ZR04] proposes increasing the number of tuples processed by each relational operator, improving instruction locality when running single-query-at-a-time DSS workloads. Unfortunately, similar techniques cannot apply to OLTP workloads because transactions typically do not form long pipelines of database operators.

Instruction locality can be improved by altering the binary code layout so that run-time code paths are as conflict-free and stored as contiguously as possible [RV+97] [RB+01]. In the example of Figure 3 one such optimization would be to place procedure A's code on address 20, so that it does not conflict with the for-loop code. Such compiler optimizations are based on static profile data collected when executing a certain targeted workload, therefore, they may hurt performance when executing other workloads. Moreover, such techniques cannot satisfy all conflicting code paths from all different execution threads.

A complementary approach is instruction prefetching in the hardware [CLM97]. Call graph prefetching [APD03] collects information about the sequence of database functions calls and prefetches the function most likely to be called next. The success of such a scheme depends on the predictability of function call sequences. Unfortunately, OLTP workloads exhibit highly unpredictable instruction streams that challenge even the most sophisticated prediction mechanisms (the evaluation of call graph prefetching is done through relatively simple DSS queries [APD03]).

3 *Steps*: Introducing cache-resident code

All OLTP transactions, regardless of the specific actions they perform, execute common database mechanisms (i.e., index traversing, buffer pool manager, lock manager, logging). In addition, OLTP typically processes hundreds of requests concurrently (the top performing system in the TPC-C benchmark suite supports over one million users and handles hundreds of concurrent client connections [TPC04]). High-performance disk subsystems and high-concurrency locking protocols ensure that, at any time, there are multiple threads in the CPU ready-to-run queue.

We propose to exploit the characteristics of OLTP code by reusing instructions in the cache across a group of transactions, effectively turning an arbitrarily large OLTP code footprint into nearly cache-resident code. We synchronize transaction groups executing common code fragments, improving performance by exploiting the high degree of OLTP concurrency. The rest of this section describes the design and implementation of *Steps,* and details its behavior using transactional microbenchmarks.

3.1 Basic implementation of *Steps*

Transactions typically invoke a basic set of operations: *begin, commit, index fetch, scan, update, insert,* and *delete.* Each of those operations involves several DBMS functions and can easily overwhelm the L1-I cache of modern processors. Experimenting with the Shore database storage manager [Ca+94] on a CPU with 64KB L1-I cache, we find that even repeated execution of a single operation always incurs additional L1-I misses. Suppose that N transactions, each being carried out by a thread, perform an *index fetch* (traverse a B-tree, lock a record, and read it). For now, we assume that transactions execute uninterrupted (all pages are in main memory and locks are granted immediately). A DBMS would execute one *index fetch* after another, incurring more L1-I cache misses with each transaction execution. We propose to reuse the instructions one transaction brings in the cache, thereby eliminating misses for the remaining N-1 transactions.

As the code path is almost the same for all N transactions (except for minor, key-value processing), *Steps* follows the code execution for one transaction and finds the point at which the L1-I cache starts evicting previously-fetched instructions. At that point *Steps* context-switches the CPU to another thread. Once that thread reaches the same point in the code as the first, we switch to the next. The Nth thread switches back to the first one, which fills the cache with new instructions. Since the last N-1 threads execute the same instructions as the first, they incur significantly fewer L1-I misses (*conflict* misses, since each code fragment's footprint is smaller than the L1-I cache).

Figures 4a and 4b illustrate the scenario mentioned above for two threads. Using *Steps*, one transaction paves the L1-I cache, incurring all compulsory misses. A second, similar transaction follows closely, finding all the instructions it needs in the cache. Next, we describe (a) how to minimize the context-switch code size, and, (b) where to insert the context-switch calls in the DBMS source code.

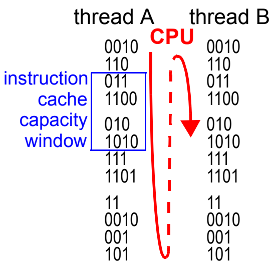


Figure 4a. *As the instruction cache cannot fit the entire code, when the CPU switches (dotted line) to thread B it will incur the same number of misses.*

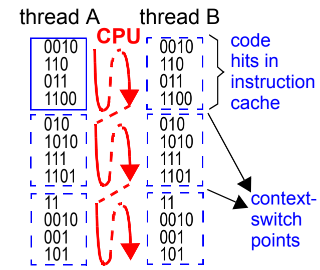


Figure 4b. *If we "break" the code into three pieces that fit in the cache, and switch execution back and forth between the two threads, thread B will find all instructions in the cache.*

3.1.1 Fast, efficient context-switching

Switching execution from one thread (or process) to another involves updating OS and DBMS software structures, as well as updating CPU registers. Thread switching is less costly than process switching (depending on the implementation). Most commercial DBMS involve a light-weight mechanism to pass on CPU control (Shore uses user-level threads). Typical context-switching mechanisms, however, occupy a significant portion of the L1-I cache and take hundreds of processor cycles to run. Shore's context-switch, for instance, occupies half of Pentium III's 16KB L1-I cache.

To minimize the overhead of context-switch we apply a universal design guideline: *make the common case fast.* The common case here is switching between transactions executing the same operation. *Steps* executes only the *core* context-switch code and updates only CPU state, ignoring thread-specific software structures such as the ready queue, until they must be updated. The minimum code needed to perform a context-switch on a IA-32 architecture — save/restore CPU registers and switch the base and stack pointers — is 48 bytes (76 in our implementation). Therefore, it only takes three 32-byte (or two 64-byte) cache blocks to store the context-switch code. One optimization that several commercial thread packages (e.g., Linux threads) make is to skip updating the floating point registers until they are actually used. For a subset of the microbenchmarks we apply a similar optimization using a flag in the core context-switch code.

3.1.2 Finding context-switching points in Shore

Given a basic set of transactional operations, we find appropriate places in the code to insert a call to *CTX (next)* (the context-switch function)*,* where *next* is a pointer to the next thread to run. *Steps* tests candidate points in the code by executing the DBMS operations (on simple, synthetic tables) and by inserting *CTX (next)* calls before or

TABLE 1: Processors used in microbenchmarks

CPU	Cache characteristics	
AMD AthlonXP	L1 I + D cache size associativity / block size	64KB + 64KB 2-way / 64 bytes
	L2 cache size	256KB
Pentium III	L1 I + D cache size associativity / block size	16KB + 16KB 4-way / 32 bytes
	L2 cache size	256KB
Simulated IA-32 (SIMFLEX)	L1 I + D cache size associativity	[16, 32, 64KB] [direct, 2, 4, 8, full]

after major function calls. Using hardware counters (available on almost all processors [INT04]), we measure the L1-I cache misses for executing various code fragments. Starting from the beginning of a DBMS operation and gradually moving towards its end, *Steps* compares the number of L1-I misses the execution of a code fragment incurs alone with the total number of misses when executing the same fragment twice (using the fast CTX call). A CTX point is inserted as soon as *Steps* detects a knee in the curve of the number of L1-I cache misses. *Steps* continues this search until it covers the entire high-level code path of a DBMS operation, for all operations.

The method of placing CTX calls described above does not depend on any assumptions about the code behavior or the cache architecture. Rather, it dynamically inspects code paths and chooses every code fragment to reside in the L1-I cache as long as possible across a group of interested transactions. If a code path is self-conflicting (given the associativity of the cache), then our method will place CTX calls around a code fragment that may have a significantly smaller footprint than the cache size, but will have fewer conflict misses when repeatedly executed. Likewise, this method also explicitly includes the context-switching code itself when deciding switching points.

The rest of this section evaluates *Steps* using microbenchmarks, whereas the complete implementation for OLTP workloads is described in Section 4. In all experiments we refer as “Shore” to the original unmodified system and as “Steps” to our system built on top of Shore.

3.2 *Steps* in practice: microbenchmarks

We conduct experiments on the processors shown in Table 1. Most experiments run on the AthlonXP, which features a large, 64KB L1-I cache. High-end installations typically run OLTP workloads on server processors (such as the ones shown in Figure 1b). In our work, however, we are primarily interested in the number of L1-cache misses. From the hardware perspective, this metric depends on the L1-I cache characteristics: size, associativity, and block size (and not on clock frequency, or the L2 cache). Moreover, L1-I cache misses are measured accurately using processor counters, whereas time-related metrics (cycles, time spent on a miss) can only be estimated and depend on the entire system configuration. Instruction misses, however, translate directly to stall time since they cannot be overlapped with out-of-order execution.

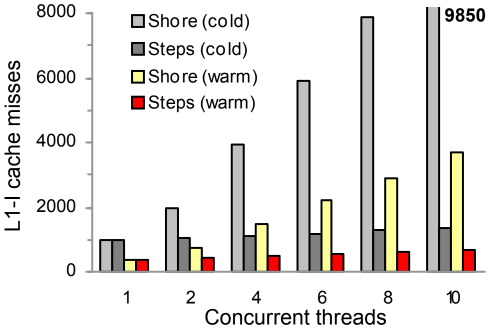


Figure 5. *Proof of concept:* Steps *reduces significantly instruction-cache misses as the group of concurrent threads increases, both with cold and with warm caches.*

Shore runs under Linux 2.4.20. We use PAPI [MB+99] and the `perfctr` library to access the AthlonXP and PIII counters. The results are based on running *index fetch* on various tables consisting of 25 `int` attributes and 100,000 rows each. The code footprint of *index fetch* without searching for the index itself (which is already loaded) is 45KB, as measured by a cache simulator (described in Section 3.2.4). Repeatedly running *index fetch* would incur no additional misses in a 45K *fully-associative* cache, but may incur conflict misses in lower-associativity caches, as explained in Figure 3. We report results averaged over 10 threads, each running *index fetch* 100 times.

3.2.1 Instruction misses and thread group size

We measure L1-I cache misses for *index fetch*, for various thread group sizes. Both *Steps* and Shore execute the fast CTX call, but *Steps* multiplexes thread execution, while Shore executes the threads serially. We first start with a cold cache and flush it between successive *index fetch* calls, and then repeat the experiment starting with a warm cache. Figure 5 shows the results on the AthlonXP.

Steps only incurs 33 misses for every additional thread, with both a cold and a warm cache. Under Shore, each additional thread adds to the total exactly the same number of misses: 985 for a cold cache (capacity misses) and 373 for a warm cache (all conflict misses since the working set of *index fetch* is 45KB). The numbers show that Shore could potentially benefit from immediately repeating the execution of the same operation across different threads. In practice, this does not happen because: (a) DBMS threads suspend and resume execution at different places of the code (performing different operations), and, (b) even if somehow two threads did synchronize, the regular context-switch code would itself conflict with the DBMS code. If the same thread, however, executes the same operation immediately, it will enjoy a warm cache. For the rest of the experiments we always warm up Shore with the same operation, and use the fast CTX call, therefore reporting worst-case lower bounds.

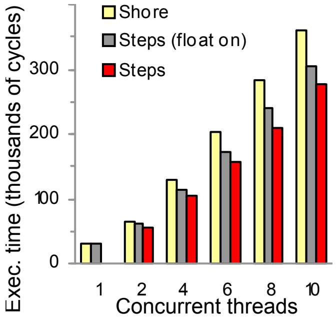


Figure 6a. *Execution time (CPU cycles) for one to ten concurrent threads.* Steps *with* float on *always updates floating point registers.*

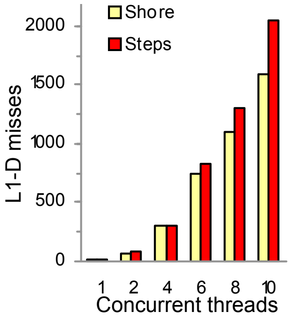


Figure 6b. *L1-D cache misses for one to ten concurrent threads.*

The following brief analysis derives a formula for the L1-I cache miss reduction bounds as a function of the thread group size (for similarly structured operations with no exceptional events). Suppose executing an operation *P* once, with cold cache, yields m_P misses. Executing *P, N* times, flushing the cache in-between, yields $N \cdot m_P$ misses. A warm cache yields $N \cdot a \cdot m_P$, $0 < a \le 1$ misses because of fewer capacity misses. In *Steps,* all threads except the first incur $N \cdot \beta \cdot m_P$ misses, where $0 < \beta < 1$. For a group size of *N*, the total number of misses is $m_P + (N-1) \cdot \beta \cdot m_P$. For an already warmed-up cache this is: $a \cdot m_P + (N-1) \cdot \beta \cdot m_P$. When comparing *Steps* to Shore, we express the miss reduction percentage as: $(1 - \#\text{misses after} / \#\text{misses before}) \cdot 100\%$. Therefore, the bounds for computing the L1-I cache miss reduction are:

$$\frac{(N-1)}{N} \cdot (1-\beta) \cdot 100\% \qquad\qquad \frac{N-1}{N} \cdot \left(1 - \frac{\beta}{a}\right) \cdot 100\%$$

for cold cache   for warm cache

N: group size

For *index fetch*, we measure $a = 0.373$, $\beta = 0.033$, giving a range of 82% - 87% of overall reduction in L1-I cache misses for 10 threads, and 90% - 96% for 100 threads. For the *tuple update* code in Shore, the corresponding parameters are: $a = 0.35$ and $\beta = 0.044$.

The next microbenchmarks examine how the savings in L1-I cache misses translate into execution time and how *Steps* affects other performance metrics.

3.2.2 Speedup and level-one data cache misses

Keeping the same setup as in 3.2.1 and providing Shore with a warmed-up cache we measure the execution time in CPU cycles and the number of level-one data (L1-D) cache misses on the AthlonXP. Figure 6a shows that *Steps* speedup increases with the number of concurrent threads. We plot both *Steps* performance with a CTX function that always updates floating point registers (float on) and with a function that skips updates. The speedup for 10 threads is 31% while for a cold cache it is 40.7% (not shown).

While a larger group promotes instruction reuse it also increases the *collective data working set*. Each thread operates on a set of private variables, buffer pool pages, and metadata which form the thread's data working set. Multiplexing thread execution at the granularity *Steps* does, results in a larger collective working set which can overwhelm the L1-D cache (when compared to Shore). Figure 6b shows that *Steps* incurs increasingly more L1-D cache misses as the thread group size increases. For up to four threads, however, the collective working set has comparable performance to single-thread execution.

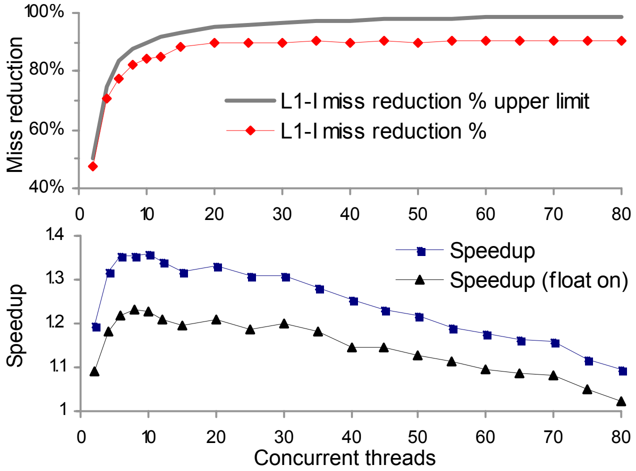


Figure 7. *Lower bounds for speedup using a warm cache for Shore (bottom graph) and percentage of reduction in L1-I cache misses (top graph) of* Steps *over Shore, for 2-80 concurrent threads. The top line shows the maximum possible reduction.*

Fortunately, L1-D cache misses have minimal effect on execution time (as also seen by the *Steps* speedup). The reason is that L1-D cache misses that hit in the L2 cache (i.e., are serviced within 5-10 cycles) can be easily overlapped by out-of-order execution [AD+99]. Moreover, in the context of Simultaneous Multithreaded Processors (SMT), it has been shown that for 8 threads executing simultaneously an OLTP workload and sharing the CPU caches, additional L1-D misses can be eliminated [LB+98].

On the other hand, there is no real incentive in increasing the group size beyond 10-20 threads, as the upper limit in the reduction of L1-I cache misses is already 90-95%. Figure 7 plots the *Steps* speedup (both with float on/off) and the percentage of L1-I cache misses reduction for 2-80 concurrent threads. The reason that the speedup deteriorates for groups larger than 10 threads is because of the AMD's small, 256KB unified L2 cache. In contrast to L1-D cache misses, L2-D misses cannot be overlapped by out-of-order execution. *Steps* always splits large groups (discussed in Section 4) to avoid the speedup degradation.

3.2.3 Detailed behavior on two different processors

The next experiment examines a wide range of changes in hardware behavior between *Steps* and Shore for *index fetch* with 10 threads. We experiment with both the Athlon XP and the Pentium III, using the same code and a CTX function that updates all registers (float optimization is off). The Pentium III features a smaller, 16KB L1-I and L1-D cache (see also table 1 for processor characteristics). Since the CTX points in Shore were chosen when running

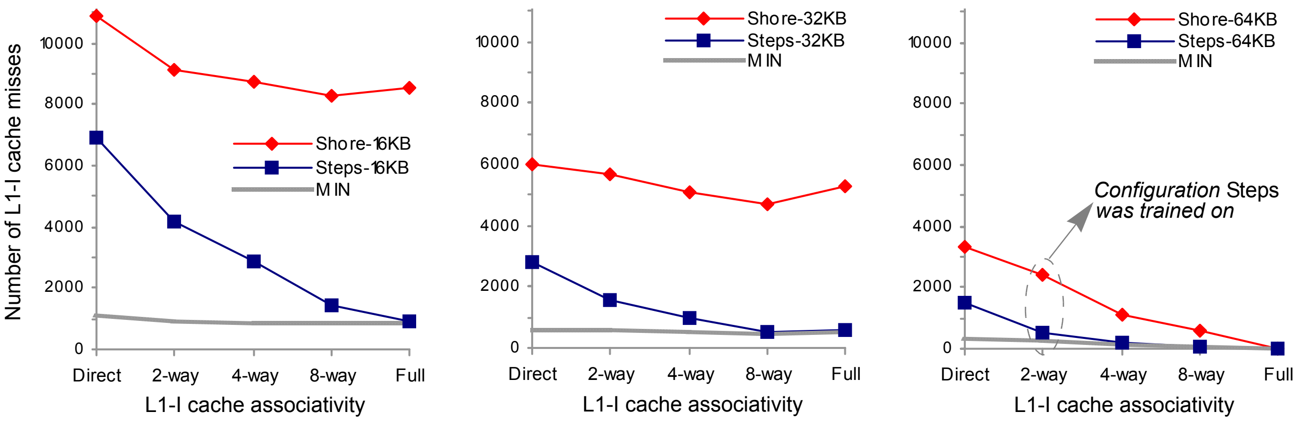


Figure 9a, 9b, 9c. *Simulation results for* index fetch *with 10 threads. We use a L1-I cache with a 64-byte cache block, varying associativity (direct, 2-, 4-, 8-way, full) and size (16KB, 32KB, 64KB).* Steps *eliminates all capacity misses and achieves up to 89% overall reduction (out of 90% max possible) in L1-I misses (max performance is for the 8-way 32KB and 64KB caches).*

on the AthlonXP (64KB L1-I cache), we expect that this version of *Steps* on the Pentium III will not be as effective in reducing L1-I cache misses as on the AthlonXP. The results are in Figure 8. Our observations for each event counted, in the order they appear in the graph, follow.

Execution time and L1-I cache misses. *Steps* is also effective on the Pentium III despite its small cache, reducing L1-I cache misses to a third (66% out of a maximum possible 90% reduction). Moreover, the speedup on the Pentium is higher than the AthlonXP, mainly because the absolute number of misses saved is higher (absolute numbers for *Steps* are on top of each bar in Figure 8). The last bar in Figure 8 shows the reduction in the cycles the processor is stalled due to lack of instructions in the cache (event only available on the Pentium III). The reduction percentage matches the L1-I cache miss reduction.

Level-one data cache. *Steps* incurs significantly more L1-D cache misses on the Pentium's small L1-D cache (109% more misses). However, the CPU can cope well by overlapping misses and perform 24% faster.

Level-two cache. L2 cache performance does not have an effect on the specific microbenchmark since almost all data and instructions can be found there. We report L2 cache performance in the next section, when running a full OLTP workload.

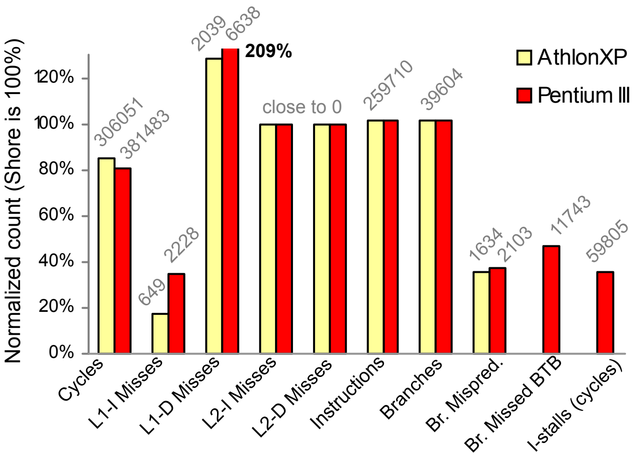


Figure 8. *Relative performance of* Steps *compared to Shore, for* index fetch *with 10 concurrent threads, on both the AthlonXP and the Pentium III. The two last bars are events exclusively available on the Pentium.*

Instructions and branches retired. As expected, *Steps* executes slightly more instructions (1.7%) and branches (1.3%) due to the extra context-switch code.

Mispredicted branches. *Steps* reduces mispredicted branches to almost a third on both CPUs (it eliminates 64% of Shore's mispredicted branches). This is an important result coming from *Steps*' ability to provide the CPU with frequently repeating execution patterns. We verify this observation via an event available to Pentium III (second to last bar in Figure 8), that shows a reduction in the number of branches missing the Branch Target Buffer (BTB), a small cache for recently executed branches.

3.2.4 Varying L1-I cache characteristics

The last microbenchmark varies L1-I cache characteristics using SIMFLEX [HS+04], a Simics-based [MC+02], full-system simulation framework developed at the Computer Architecture Lab of Carnegie Mellon. We use Simics/SIMFLEX to emulate a x86 processor (Pentium III) and associated peripheral devices (using the same setup as in the real Pentium). Simics boots and runs the exact same binary code of Linux and the Shore/*Steps* microbenchmark, as in the real machines. Using SIMFLEX's cache component we modify the L1-I cache characteristics (size, associativity, block size) and run the 10-thread *index fetch* benchmark. The reported L1-I cache misses are exactly the same as in a real machine with the same cache characteristics. Metrics in simulation involving timing are subject to assumptions made by programmers and cannot possibly match real execution times. Figures 9a, 9b, and 9c show the results for a fixed 64-byte cache block size, varying associativity for a 16KB, 32KB, and 64KB L1-I cache.

As expected, increasing the associativity reduces instruction conflict misses (except for a slight increase for fully-associative 16KB and 32KB caches, due to the LRU replacement policy resulting in more capacity misses).

The conflict miss reduction for *Steps* is more dramatic in a small cache (16KB). The reason is that with a 45KB working set for *index fetch* even a few CTX calls can eliminate all capacity misses for the small caches. Since *Steps* is trained on a 2-way 64KB cache, smaller caches with the same associativity incur more conflict misses. As the associativity increases those additional L1-I misses disappear. Despite a fixed training on a large cache, *Steps* performs very well across a wide range of cache architectures, achieving a 89% overall reduction in L1-I misses — out of 90% max possible — for the 8-way 32KB and 64KB caches. Experiments with different cache block sizes (not shown here) find that larger blocks further reduce L1-I misses, in agreement with the results in [RG+98].

4 Applying *Steps* to OLTP workloads

So far we saw how to efficiently multiplex the execution of concurrent threads running the same transactional DBMS operation when (a) those threads run uninterrupted, and (b) the DBMS does not schedule any other threads. This section removes all previous assumptions and describes how *Steps* works in full-system operation. The design goal is to take advantage of the fast CTX calls and maintain high concurrency for similarly structured operations in the presence of locking, latching (which provides exclusive access to DBMS structures), disk I/O, aborts and roll-backs, and other concurrent system operations (e.g., deadlock detection, buffer pool page flushing).

The rest of this section describes the full *Steps* implementation (Section 4.1), presents the experimentation setup (4.2) and the TPC-C results (4.3), and briefly discusses applicability in commercial DBMS (4.4).

4.1 Full *Steps* implementation

Steps employs a two-level transaction synchronization mechanism. At the higher level, all transactions about to perform a single DBMS operation form execution teams. We call S-threads all threads participating in an execution team (excluding system-specific threads or processes and threads which are blocked for any reason). Once all S-threads belong to a team, the CPU proceeds with the lower-level transaction synchronization scheme within a single team, following a similar execution schedule as in the previous section. Next, we detail synchronization mechanisms (Section 4.1.1), different code paths (Section 4.1.2), and threads leaving their teams (Section 4.1.3). Section 4.1.4 summarizes the changes to Shore code.

4.1.1 Forming and scheduling execution teams

To facilitate a flexible assignment of threads to execution teams and construct an efficient CPU schedule during the per-team synchronization phase, each DBMS operation is associated with a double-linked list (Figure 10). S-threads are part of such a list (depending on which operation they are currently executing), while all other threads have the `prev` and `next` pointers set to zero. The list for each execution team guides the CPU scheduling decisions. At each CTX point the CPU simply switches to the next thread in the list. S-threads may leave a team (disconnect) for several reasons. Transactions give up (yield) the CPU when they (a) block trying to acquire an exclusive lock (or access an exclusive resource), or on an I/O request, and, (b) when they voluntarily yield control as part of the code logic. We call *stray* the threads that leave a team.

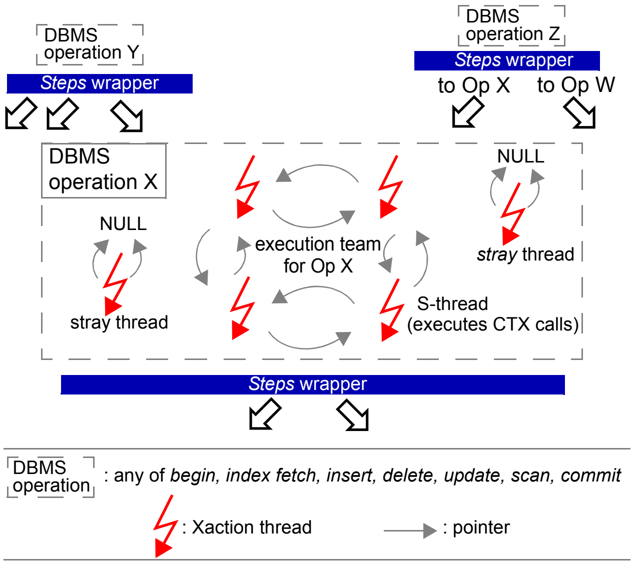


Figure 10. *Additions to the DBMS code: Threads are associated with list nodes and form per-operation lists, during the* Steps *setup code at the end of each DBMS operation.*

The code responsible for team formation is a thin wrapper that runs every time a transaction finishes a single DBMS operation ("*Steps* wrapper" in Figure 10). It disconnects the S-thread from the current list (if not stray) and connects it to the next list, according to the transaction code logic. If a list reaches the maximum number of threads allowed for a team (a user-defined variable), then the transaction will join a new team after the current team finishes execution. Before choosing the next team to run, all stray threads are given a chance to join their respective teams (next DBMS operation on their associated transaction's code logic). Finally, the *Steps* wrapper updates internal statistics, checks with the system scheduler if other tasks need to run, and picks the next team to run[2].

Within each execution team *Steps* works in a "best-effort" mode. Every time a transaction (or any thread) encounters a CTX point in the code, it first checks if it is an S-thread and then passes the CPU to the next thread in the list. All S-threads in the list eventually complete the current DBMS operation, executing in a round-robin fash-

[2] Different per-team scheduling policies may apply at this point. In our experiments, picking the next operation that the last member of a list (or the last stray thread) is interested, worked well in practice since the system scheduler makes sure that every thread makes progress.

TABLE 2: Operation classification for overlapped code

DBMS operation	cross-transaction code overlap		
	always	same tables	same tables + split Op
begin / commit	✓		
fetch		✓	
insert			✓
delete			✓
update	✓		
scan	✓		

ion, the same way as in Section 3. This approach does not explicitly provide any guarantees that all threads will remain synchronized for the duration of the DBMS operation. It provides, however, a very fast context-switching mechanism during full-system operation (the same list-based mechanism was used in all microbenchmarks). If all threads execute the same code path without blocking, then *Steps* will achieve the same L1-I cache miss reduction as in the previous section. Significantly different code paths across transactions executing the same operation or exceptional events that cause threads to become stray may lead to reduced benefits in the L1-I cache performance. Fortunately, we can reduce the effect of different code paths (Section 4.1.2) and exceptional events (4.1.3).

4.1.2 Maximizing code overlap across transactions

If an S-thread follows a significantly different code path than other threads in its team (e.g., traverse a B-tree with fewer levels), the assumed synchronization breaks down. That thread will keep evicting useful instructions with code that no one else needs. If a thread, however, exits the current operation prematurely (e.g., a key was not found), the only effect will be a reduced team size, since the thread will wait to join another team. To minimize the effect of different code paths we follow the next two guidelines:

1. Have a separate list for each operation that manipulates a different index (i.e., *index fetch (table1), index fetch (table2),* and so on).

2. If the workload does not yield high concurrency for similarly structured operations, we consider defining finer-grain operations. For example, instead of an *insert* operation, we can maintain a different list for creating a record and a different one for updating an index.

Table 2 shows all transactional operations along with their degree of cross-transaction overlapped code. *Begin, commit, scan,* and *update* are independent of the database structure and use a single list each. *Index fetch* code follows different branches depending on the B-tree depth, therefore a separate list per index maximizes code overlap. Lastly, *insert* and *delete* code paths may differ across transactions even for same indices, therefore it may be necessary to define finer-grain operations. While experimenting with TPC-C we find that following only the first guideline (declaring lists per index) is sufficient. Small variations in the code path are unavoidable (e.g., utilizing a different attribute set or manipulating different strings) but the main function calls to the DBMS engine are generally the same across different transactions. For workloads with an excessive number of indices, we can use statistics collected by *Steps* on the average execution team size per index, and consolidate teams from different indices. This way *Steps* trades code overlap for an increased team size.

4.1.3 Dealing with stray transactions

S-threads turn into stray when they block or voluntarily yield the CPU. In *preemptive* thread packages the CPU scheduler may also preempt a thread after its time quantum has elapsed. The latter is a rare event for *Steps* since it performs switches at orders of magnitude faster times than the quantum length. In our implementation on Shore we modify the thread package and intercept the entrance of `block` and `yield` to perform the following actions:

1. Disconnect the S-thread from the current list.
2. Turn the thread into stray, by setting pointers `prev` and `next` to zero. Stray threads bypass subsequent CTX calls and fall under the authority of the regular scheduler. They remain stray until they join the next list.
3. Update all thread package structures that were not updated during the fast CTX calls. In Shore these are the current running thread, and the ready queue status.
4. Pass a hint to the regular scheduler that the next thread to run should be the next in the current list (unless a system or a higher priority thread needs to run first).
5. Give up the CPU using regular context-switching.

Except for I/O requests and non-granted locks, transactions may go astray because of mutually exclusive code paths. Frequently, a database programmer protects accesses or modifications to a shared data structure by using a mutex (or a latch). If an S-thread calls CTX while still holding the mutex, all other threads in the same team will go astray as they will not be able to access the protected data. If the current operation's remaining code (after the mutex release) can still be shared, it may be preferable to skip the badly placed CTX call. This way *Steps* only suffers momentarily the extra misses associated with executing a small, self-evicting piece of code.

Erasing CTX calls is not a good idea since the specific CTX call may also be accessed from different code paths (for example, through other operations) which do not necessarily go through acquiring a mutex. *Steps* associates with every thread a counter that increases every time the thread acquires a mutex and decreases when releasing it. Each CTX call tests if the counter is non-zero in which case it lets the current thread continue running without giving up the CPU. In Shore, there were only two places in the code that the counter would be non-zero.

TABLE 3: System configuration

CPU	AthlonXP, 2GB RAM, Linux 2.4.20
Storage	one 120GB main disk, one 30GB log disk
Buffer pool size	Up to 2GB
Page size	8192 Bytes
Shore locking hierarchy	Record, page, table, entire database
Shore locking protocol	Two phase locking

4.1.4 Summary of changes to the DBMS code

The list of additions and modifications to the Shore code base is the following. We added the wrapper code to synchronize threads between calls to DBMS operations (*Steps* wrapper, 150 lines of C++), the code to perform fast context-switching (20 lines of inline assembly), and we also added global variables for the list pointers representing each DBMS operation. We modified the thread package code to update the list nodes properly and thread status whenever blocking, yielding, or changing thread priorities (added/changed 140 lines of code). Finally, we inserted calls to our custom CTX function into the source code (as those were found during the microbenchmarking phase). Next, we describe the experimentation testbed.

4.2 Experimentation setup

We experiment with the TPC-C benchmark which models a wholesale parts supplier operating out of a number of warehouses and their associated sales districts [TPC04]. It involves a mix of five different types of transactions. The two most frequently executed transactions (88% of the time) are the New Order and the Payment transactions. TPC-C transactions operate on nine tables and they are all based on the DBMS operations listed in Table 2.

The TPC-C toolkit for Shore is written at CMU. Table 3 shows the basic configuration characteristics of our system. To ensure high concurrency and reduce the I/O bottleneck in our two-disk system we cache the database in the buffer pool and allow transactions to commit without waiting for the log to be flushed on disk (the log is flushed asynchronously). A reduced buffer pool size would cause I/O contention allowing only very few threads to be runnable at any time. High-end installations can hide the I/O latency by parallelizing requests on multiple disks. To mimic a high-end system's CPU utilization, we set user thinking time to zero and keep the standard TPC-C scaling factor (10 users per Warehouse), essentially having as many concurrent threads as the number of users. We found that, when comparing *Steps* with Shore running New Order, *Steps* was more efficient in inserting multiple subsequent records on behalf of a transaction (because of a slot allocation mechanism that was avoiding overheads when inserts were spread across many transactions). We modified slightly New Order by removing one insert from inside a for-loop (but kept the remaining inserts).

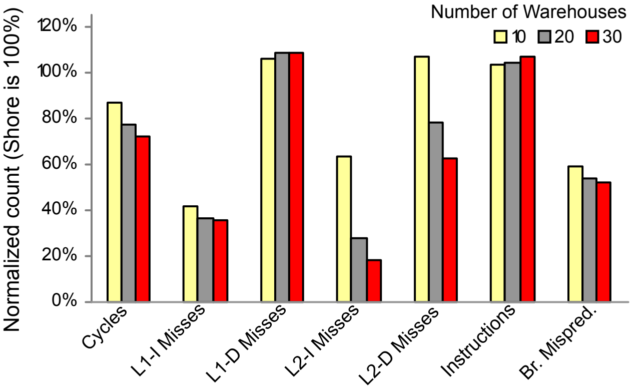


Figure 10. *Transaction mix includes only the Payment transaction, for 10-30 Warehouses (100-300 threads).*

For all experiments we warm up the buffer pool and measure CPU events in full-system operation, including background I/O processes that are not optimized using *Steps*. Measurement periods range from 10sec - 1min depending on the time needed to complete a pre-specified number of transactions. All reported numbers are consistent across different runs, since the aggregation period is large in terms of CPU time. Our primary metric is the number of L1-I cache misses as it is not affected by the AthlonXP's small L2 cache (when compared to server processors shown in Figure 1b).

***Steps* setup:** We keep the same CTX calls used in the microbenchmarks but without using floating point optimizations, and without re-training *Steps* on TPC-C indexes or tables. Furthermore, we refrain from using *Steps* on the TPC-C application code. Our goal is to show that *Steps* is workload-independent and report lower bounds for performance metrics by not using optimized CTX calls. We assign a separate thread list to each *index fetch, insert,* and *delete* operating on different tables while keeping one list for each of the rest operations. Restricting execution team sizes has no effect since in our configuration the number of runnable threads is low. For larger setups, *Steps* can be configured to restrict team sizes, essentially creating multiple independent teams per DBMS operation.

4.3 TPC-C results

Initially we run all TPC-C transaction types by themselves varying the database size (and number of users). Figure 10 shows the relative performance of *Steps* over Shore when running the Payment transaction with standard TPC-C scaling for 10, 20, and 30 warehouses. The measured events are: execution time in CPU cycles, cache misses for both L1 and L2 caches, the number of instructions executed, and the number of mispredicted branches. Results for other transaction types were similar. *Steps* outperforms Shore, achieving a 60-65% reduction in L1-I cache misses, a 41-45% reduction in mispredicted branches, and a 16-39% speedup (with no floating point optimizations). The benefits increase as the database size (and number of

TABLE 4: Team sizes per DBMS operation in Payment

Warehouses →	10		20		30	
Operation (table)↓	in	out	in	out	in	out
index fetch (C)	8.6	8.6	16	16	25.2	24.7
index fetch (D)	8.9	1.7	16.2	2.6	31.7	5.3
index fetch (W)	8.9	0.5	16.6	1	30	1.9
scan (C)	9.4	8.2	16	14.3	26.2	23.7
insert (H)	7.9	7.8	14.9	14.6	24	23.2
update (C, D, W)	7.5	7.2	14	12.3	21.6	19
average team size	8.6	6.9	15.9	12.3	26.4	20.4
# of ready threads	15		28		48.4	

users) scale up. The increase in L1-D cache misses is marginal. *Steps* speedup is also fueled by fewer L2-I and L2-D misses as the database size increases. *Steps* makes better utilization of AMD's small L2 cache as fewer L1-I cache misses also translate into more usable space in L2 for data.

Table 4 shows for each configuration (10, 20, and 30 warehouses running Payment) how many threads on average enter an execution team for a DBMS operation and exit without being strays, along with how many threads are ready to run at any time and the average team size. The single capital letters in every operation correspond to the TPC-C tables/indices used (Customer, District, Warehouse, and History). *Steps* is able to group on average half of the available threads. Most of the operations yield a low rate for producing strays, except for *index fetch* on District and Warehouse. In small TPC-C configurations, exclusive locks on those tables restrict concurrency.

Next, we run the standard TPC-C mix, excluding the non-interactive Delivery transaction (TPC-C specifies up to 80sec queueing delay before executing Delivery). Figure 11 shows that the four-transaction mix follows the general behavior of the Payment mix, with the reduction in instruction cache misses (both L1 and L2) being slightly worse. Statistics for the team sizes reveal that this configuration forces a smaller average team size due to the increased number of unique operations. For the 10-warehouse configuration, there are 14 ready threads, and on average, 4.3 threads exit from a list without being stray. Still, this means a theoretical bound of a 77% reduction in L1-I cache misses, and *Steps* achieves a 56% reduction while handling a full TPC-C workload and without being optimized for it specifically. Results for different mixes of TPC-C transactions were similar.

4.4 Applicability

Steps has the following two attractive features that simplify integration in a commercial DBMS: (a) its application is incremental as it can target specific operations and co-exist with other workloads, (e.g., DSS, which can simply bypass CTX calls), and (b) the required code modifications are restricted to a very specific small subset of the code, the thread package. Most commercial thread packages implement preemptive threads. As a result, DBMS code is *thread safe*, meaning that programmers develop DBMS code anticipating random context-switches that can occur at any time. This is also true for DBMS using processes instead of threads (such as Oracle on Linux). Thread safe code ensures that any placement of CTX calls throughout the code will not break any assumptions.

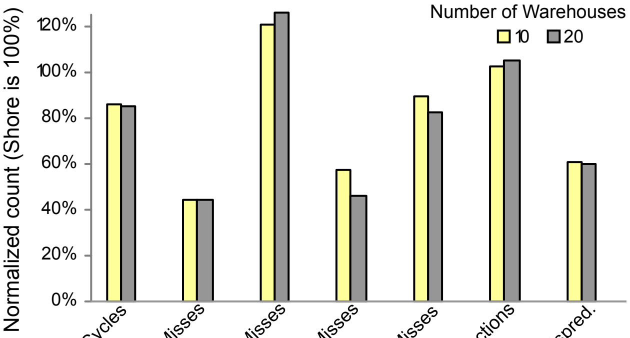


Figure 11. *Transaction mix includes all transactions except the non-interactive Delivery transaction, for 10-20 Warehouses.*

To apply *Steps* to any thread-based DBMS the programming team needs to augment the thread package to support fast context-switching. Process-based systems may require changes to a larger subset of the underlying OS code. In general, a *Steps* CTX call should bypass the OS scheduler and update only the absolute minimum state needed by a different thread/process for code execution that does not give up CPU control. Whenever a thread gives up CPU control through a mechanism different than fast CTX, all state needed before invoking a regular context-switch should be updated accordingly. The next phase is to add the *Steps* wrapper in each major DBMS operation. This thin wrapper provides the high-level, per-operation transaction synchronization used in *Steps*.

The final phase is to place CTX calls in the code depending on the underlying cache architecture. Readily available binary tools can automate CTX call placement (using either trace-based cache simulation or CPU hardware counters). For example, `valgrind/cachegrind` is a cache profiling tool which can be used to track all instruction cache misses during the execution of a sample operation. It can be easily configured to output the code lines where CTX calls should be placed using a simple cache simulator. Next, a binary modification tool [SE94] can be used to insert the CTX calls to the DBMS binary, while a race-detection binary tool [SB+97] can be used to pinpoint badly placed CTX calls which may cause races or force S-threads to go astray. Moreover, a compiler can "color" the L1-I cache blocks containing the CTX code to make them permanently reside in the cache, thereby reducing conflict misses. The only workload-specific tuning required is the creation of per-index execution teams, which can be done once the database schema is known.

Epilogue

This paper demonstrates that the key to optimal performance for OLTP workloads on modern processors is within the DBMS design. Computer architects have already pushed next-generation processor designs to the market and are now working on proactive memory systems to eliminate data cache misses. Instruction cache stalls are a major barrier towards bringing OLTP performance on par with scientific and engineering workload performance. Software vendors have the best insight of how and where to affect software behavior. *Steps*, our proposed technique, is orthogonal to compiler techniques and its benefits are always additional to any binary-optimized configuration. In this paper we show that *Steps* minimizes both capacity and conflict instruction cache misses of OLTP with arbitrary long code paths, without increasing the size or the associativity of the instruction cache.

Acknowledgements

We thank Babak Falsafi, James Hamilton, Bruce Lindsay, Ken Ross, Michael Stonebraker, and the reviewers for their comments. We also thank Nikos Hardavellas and Tom Wenisch for SimFlex, and Mengzhi Wang for the Shore TPC-C toolkit. This work is supported in part by an IBM faculty partnership award and by NSF grants CCR-0113660, IIS-0133686, and CCR-0205544.

References

[AD+01] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. "Weaving Relations for Cache Performance." In *Proc. VLDB*, 2001.

[AD+99] A. Ailamaki, D. J. DeWitt, at al. "DBMSs on a modern processor: Where does time go?" In *Proc. VLDB*, 1999.

[APD03] M. Annavaram, J. M. Patel, and E. S. Davidson. "Call graph prefetching for database applications." In *ACM Transactions on Computer Systems,* 21(4):412-444, November 2003.

[BGB98] L. A. Barroso, K. Gharachorloo, and E. Bugnion. "Memory System Characterization of Commercial Workloads." In *Proc. ISCA*, 1998.

[Ca+94] M. Carey et al. "Shoring Up Persistent Applications." In *Proc. SIGMOD*, 1994.

[CLM97] I-C. Chen, C-C. Lee, and T. Mudge. "Instruction prefetching using branch predition information." In *Proc. International Conference on Computer Design* 1997.

[CGM01] S. Chen, P. B. Gibbons, and T. C. Mowry. "Improving Index Performance through Prefetching." In *Proc. SIGMOD*, 2001.

[GL01] G. Graefe and P. Larson. "B-Tree Indexes and CPU Caches." In *Proc. ICDE*, 2001.

[Gra93] J. Gray. "The benchmark handbook for transaction processing systems." 2nd ed., Morgan-Kaufmann, 1993.

[HP96] J. L. Hennessy and D. A. Patterson. "Computer Architecture: A Quantitative Approach." 2nd ed, Morgan-Kaufmann, 1996.

[HS+04] N. Hardavellas, S. Somogyi, et al. "SimFlex: a Fast, Accurate, Flexible Full-System Simulation Framework for Performance Evaluation of Server Architecture." *SIGMETRICS Performance Evaluation Review, Vol. 31, No. 4, pp. 31-35,* April 2004.

[JK99] J. Jayasimha and A. Kumar. "Thread-based Cache Analysis of a Modified TPC-C Workload." In *2nd CAECW Workshop*, 1999.

[INT04] Intel Corporation. "IA-32 Intel® Architecture Software Developer's Manual, Volume 3: System Programming Guide." (*Order Number 253668*).

[KP+98] 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 *Proc. ISCA-25*, 1998.

[LB+98] J. Lo, L. A. Barroso, S. Eggers, et al. "An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors." In *Proc. ISCA-25*, 1998.

[MC+02] P. S. Magnusson, et al. "Simics: A Full System Simulation Platform." In *IEEE Computer*, 35(2):50–58, February 2002.

[MDO94] A. M. G. Maynard, C. M. Donelly, and B. R. Olszewski. "Contrasting Characteristics and Cache Performance of Technical and Multi-user Commercial Workloads." In *Proc. ASPLOS-6*, 1994.

[MB+99] P. J. Mucci, S. Browne, et al. "PAPI: A Portable Interface to Hardware Performance Counters." In *Proc. Dept. of Defense HPCMP Users Group Conference*, Monterey, CA, June 7-10, 1999.

[PM+01] S. Padmanabhan, T. Malkemus, R. Agarwal, A. Jhingran. "Block Oriented Processing of Relational Database Operations in Modern Computer Architectures." In *Proc. ICDE*, 2001.

[RB+01] A. Ramirez, L. A. Barroso, et al. "Code Layout Optimizations for Transaction Processing Workloads." In *ISCA-28*, 2001.

[RG+98] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. "Performance of database workloads on shared-memory systems with out-of-order processors." In *Proc. ASPLOS*, 1998.

[RV+97] T. Romer, G. Voelker, et al. "Instrumentation and Optimization of Win32/Intel Executables Using Etch." In *Proc. Usenix NT Workshop*, 1997.

[RB+95] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. "The Impact of Architectural Trends on Operating System Performance." In *Proc. SOSP-15*, pp.285-298, 1995.

[SB+97] S. Savage, M. Burrows, et al. "Eraser: A Dynamic Data Race Detector for Multithreaded Programs." In *ACM TOCS, vol. 15, no. 4, pp. 391-411,* November 1997.

[SA04] M. Shao and A. Ailamaki. "DBmbench: Fast and Accurate Database Workload Representation on Modern Microarchitecture." *In submission. Available as Technical Report CMU-CS-03-161.*

[SKN94] A. Shatdal, C. Kant, and J. Naughton. "Cache Conscious Algorithms for Relational Query Processing." In *Proc. VLDB*, 1994.

[SE94] A. Srivastava and A. Eustace. "ATOM: A system for building customized program analysis tools." In *Proc. SIGPLAN*, 1994.

[SBG02] R. Stets, L. A. Barroso, and K. Gharachorloo. "Detailed Comparison of Two Transaction Processing Workloads," In *Proc. 5th CAECW Workshop*, 2002.

[TPC04] Transaction Processing Performance Council. `http://www.tpc.org`

[ZR04] J. Zhou and K. A. Ross. "Buffering Database Operations for Enhanced Instruction Cache Performance." In *SIGMOD*, 2004.