Improving Instruction Cache Performance in OLTP

STAVROS HARIZOPOULOS
MIT CSAIL
and
ANASTASSIA AILAMAKI
Carnegie Mellon University

Instruction-cache misses account for up to 40% of execution time in online transaction processing (OLTP) database workloads. In contrast to data cache misses, instruction misses cannot be overlapped with out-of-order execution. Chip design limitations do not allow increases in the size or associativity of instruction caches that would help reduce misses. On the contrary, the effective instruction cache size is expected to further decrease with the adoption of multicore and multithreading chip designs (multiple on-chip processor cores and multiple simultaneous threads per core). Different concurrent database threads, however, execute similar instruction sequences over their lifetime, too long to be captured and exploited in hardware. The challenge, from a software designer's point of view, is to identify and exploit common code paths across threads executing arbitrary operations, thereby eliminating extraneous instruction misses.

In this article, we describe *Synchronized Threads through Explicit Processor Scheduling* (STEPS), a methodology and tool to increase instruction locality in database servers executing transaction processing workloads. STEPS works at two levels to increase reusability of instructions brought in the cache. At a higher level, synchronization barriers form *teams* of threads that execute the same system component. Within a team, STEPS schedules special fast context-switches at very fine granularity to reuse sets of instructions across team members. To find points in the code where context-switches should occur, we develop *autoSTEPS*, a code profiling tool that runs directly on the DBMS binary. STEPS can minimize both capacity and conflict instruction cache misses for arbitrarily long code paths.

We demonstrate the effectiveness of our approach on *Shore*, a research prototype database system shown to be governed by similar bottlenecks as commercial systems. Using microbenchmarks on real and simulated processors, we observe that STEPS eliminates up to 96% of instruction-cache misses for each additional team thread and at the same time eliminates up to 64% of mispredicted branches by providing a repetitive execution pattern to the processor. When performing a full-system evaluation on real hardware using TPC-C, the industry-standard transactional

This research was supported in part by the National Science Foundation under grants CCR-0113660, IIS-0133686, and CCR-0205544, and by an IBM Ph.D. fellowship.

Authors' addresses: S. Harizopoulos, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA; email: stavros@csail.mit.edu; A. Ailamaki, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213; email: natassa@cmu.edu.

benchmark, STEPS eliminates two-thirds of instruction-cache misses and provides up to 1.4 overall speedup.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Transaction processing; concurrency*; D.4.1 [**Operating Systems**]: Process Management—*Threads, scheduling*

General Terms: Performance, Design

Additional Key Words and Phrases: Instruction cache, cache misses

1. INTRODUCTION

Good instruction-cache performance is crucial in modern database management system (DBMS) installations running online transaction processing (OLTP) applications. Examples of OLTP applications are banking, e-commerce, reservation systems, and inventory management. A common requirement is the ability of the DBMS software to execute efficiently multiple concurrent transactions which are typically short in duration. In large-scale installations, server software leverages increasingly higher-capacity main memories and highly parallel storage subsystems to efficiently hide I/O latencies. As a result, DBMS software performance is largely determined by the ability of the processors to continuously execute instructions without *stalling*. Recent studies have shown that between 22% and 41% of the execution time in TPC-C, the prevailing OLTP benchmark, is attributed to instruction-cache misses [Shao et al. 2005; Keeton et al. 1998; Barroso et al. 1998].

Over the past few years, a wide body of research has proposed techniques to identify and reduce CPU performance bottlenecks in database workloads [Shatdal et al. 1994; Graefe and Larson 2001]. Since memory access times improve much slower than processor speed, performance is bound by instruction and data cache misses that cause expensive main-memory accesses [Ranganathan et al. 1998; Ailamaki et al. 1999]. Related research has proposed hardware and compiler techniques to address instruction-cache performance [Ranganathan et al. 1998; Ramirez et al. 2001]. The focus, however, was on single-thread execution (single transaction). While the proposed techniques can enhance instruction-cache performance by increasing *spatial locality* (utilization of instructions contained in a cache block), they cannot increase *temporal locality* (reusability of an entire instruction-cache block) since the latter is a direct function of the application nature. A recent study on Oracle reported a 556-kB OLTP code footprint [Lo et al. 1998]; with modern CPUs having 16-64-kB instruction cache sizes (L1-I cache), OLTP code paths are too long to achieve cache-residency.

While hardware or compiler techniques cannot increase instruction reusability (temporal locality) within a single execution thread, an appropriate software mechanism can potentially increase instruction reusability by exploiting common instructions in the cache *across* different concurrent threads. The payoff in such a software approach can be large, since DBMS software typically handles multiple concurrent transactions by assigning them to different threads, and, at the same time, complementary to any hardware/compiler technique. To illustrate how this idea might work, consider a thread scheduling mechanism that, as a first step, identifies 10 concurrent threads that are about to execute the same function call, the code path of which is far larger than the instruction cache. Each thread execution will yield a default number of instruction misses. If, however, we perfectly reuse the instructions one thread brings gradually in the cache, across all 10 threads, we can achieve a 90% overall reduction in instruction cache misses. That is, in a group of threads, we could potentially turn one thread's instruction cache misses to cache hits for all the other threads. Applying this idea, however, to a DBMS consisting of millions lines of code, executing different types of transactions, each with different requirements and unpredictable execution sequences (due to lock requests, frequent critical sections, I/O requests), is a challenging task.

This article reviews *Synchronized Threads through Explicit Processor Scheduling* (STEPS) [Harizopoulos and Ailamaki 2004], a methodology and its application for increasing instruction locality in database servers executing transaction processing workloads. STEPS works at two levels to increase reusability of instructions brought in the cache. At a higher level, synchronization barriers form *teams* of threads that execute the same system component. Within a team, STEPS schedules special, fast context-switches at very fine granularity, to reuse sets of instructions across team members. Both team formation and fine-grain context-switching are low-overhead software mechanisms, designed to coexist with all DBMS internal mechanisms (locking, logging, deadlock detection, buffer pool manager, I/O subsystem), and flexible enough to work with any arbitrary OLTP workload. STEPS requires only few code changes, targeted at the thread package.

In this article we also introduce and evaluate autoSTEPS, a code profiling tool that automates the application of STEPS for any database system. autoSTEPS runs directly on the DBMS software binary and can find the points in the DBMS core code where context-switches should occur, thus eliminating the need to examine and modify DBMS source code. Moreover, autoSTEPS does not depend on the cache characteristics of the machine used for profiling, and therefore, it can be configured to produce output for any arbitrary targeted cache architecture.

We demonstrate the effectiveness of our approach on *Shore* [Carey et al. 1994], a research prototype database system shown to be governed by bottlenecks similar to those of commercial systems [Ailamaki et al. 2001b]. First, using microbenchmarks on real and simulated processors, we show that STEPS eliminates up to 96% of instruction-cache misses for each additional team thread, and at the same time eliminates up to 64% of mispredicted branches by providing a repetitive execution pattern to the CPU. When performing a full-system evaluation, on real hardware, with TPC-C, the industry-standard transactional benchmark, STEPS eliminates two-thirds of instruction-cache misses and provides up to 1.4 overall speedup. To the best of our knowledge, this is the first software approach to provide explicit thread scheduling for improving instruction cache performance. The contributions of the article are

—a novel technique that enables thread scheduling at very fine granularity to reuse instructions in the cache across concurrent threads;

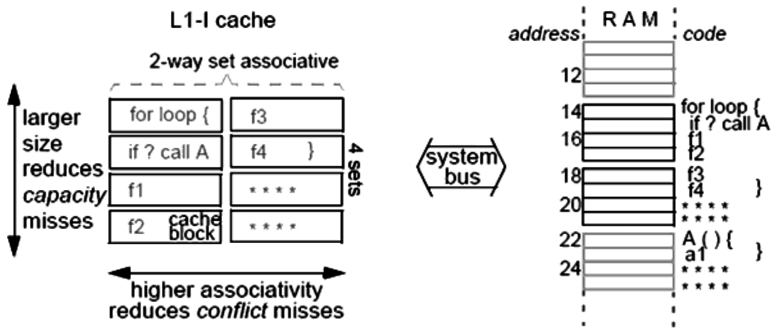


Fig. 1. Example of a two-way set associative, four-set (eight cache blocks) L1-I cache. Code stored in RAM maps to one set of cache blocks and is stored to any of the two blocks in that set. For simplicity, we omit L2/L3 caches. In this example, the *for-loop* code fits in the cache only if procedure 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.

—a tool to automatically find the points in the code that the instruction cache fills up;
—the implementation and evaluation of the presented techniques inside a research prototype database system running a full-blown multiuser transactional benchmark on real hardware.

The rest of the article is organized as follows. Section 2 contains background information on instruction caches and discusses trends in processor design along with related research efforts. Section 3 introduces STEPS and evaluates the basic implementation through microbenchmarks on three processors (AthlonXP, Pentium III, and Pentium 4) and on several different simulated cache configurations. Section 4 describes the full STEPS implementation, capable of handling any arbitrary OLTP workload and presents TPC-C results. Section 5 presents autoSTEPS and discusses its applicability to commercial DBMS software. Section 6 concludes and discusses how STEPS can be extended to apply to current and upcoming multithreaded processor designs.

2. BACKGROUND AND RELATED WORK

To bridge the CPU/memory performance gap [Hennessy and Patterson 1996], today's processors employ a hierarchy of caches that maintain recently referenced instructions and data close to the processor. Figure 1 shows an example of an instruction cache organization and explains the difference between *capacity* and *conflict* cache misses. Recent processors—for example, IBM's Power5—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., 16–64 kB), but operate at (or close to) 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.,

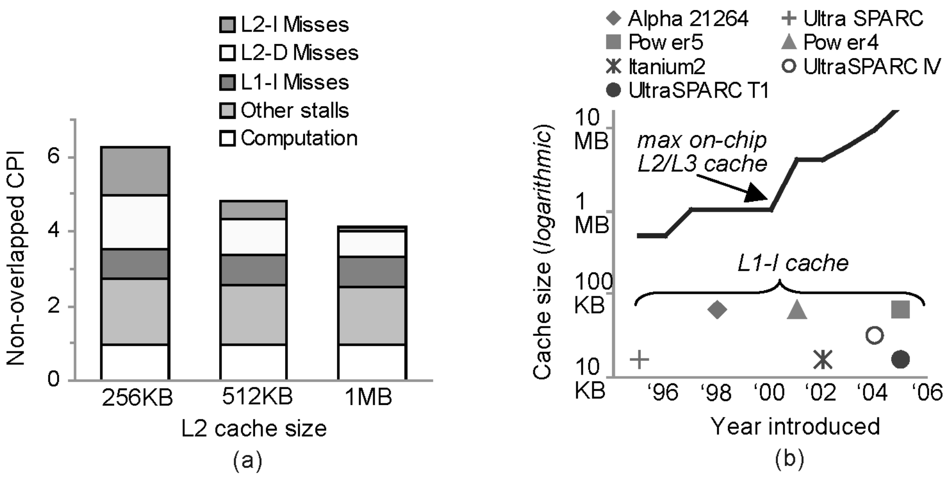


Fig. 2. (a) TPC-C CPU stall breakdown on PentiumPro, shown with non-overlapping components. With larger L2 cache size L1-I misses become the dominant stall factor (third box from top) [Keeton et al. 1998]. (b) A decade-spanning trend shows that L1-I caches do not grow, while secondary, on-chip caches become increasingly larger.

512 kB–8 MB). L2 lookup, however, is still several orders of magnitude faster than main memory access (which typically takes 300–400 cycles). Therefore, the effectiveness of cache hierarchy is extremely important for performance.

2.1 The Instruction Cache Problem

In contrast to data cache accesses, instruction cache accesses are serialized and cannot be overlapped. Instruction cache misses prevent the flow of instructions through the processor and directly affect performance. To maximize first-level instruction 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 a 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 [Keeton et al. 1998]. A recent study on Oracle has reported a 556-kB OLTP code footprint [Lo et al. 1998]. With modern CPUs having 16- to 64-kB 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, as shown in Figure 2(a) (stalls shown as nonoverlapping components; I-cache stalls are actually 41% of the total execution time [Keeton et al. 1998]). As a large (or highly associative) L1-I cache may adversely impact the CPU's clock frequency, chip designers cannot increase L1-I sizes (and/or associativity) despite the growth in secondary caches, as shown in Figure 2(b).

Current chip design trends toward improving process performance are leading to thread-parallel architectures, where multiple threads or processes can run simultaneously on a single chip via multiple on-chip processor cores (chip multiprocessors, or CMP) and/or multiple simultaneous threads per processor core (simultaneous multithreading, or SMT).[1] To be able to fit more cores on a single chip without overheating, and also save time in hardware verification, chip designers are expected to use simpler, "lean" cores as building blocks (this is exactly the philosophy behind Sun's UltraSPARC T1, which uses up to eight cores on a single chip with four threads per core). The instruction cache size of these cores is not expected to grow (for example, the UltraSPARC T1 features a 16-kB L1-I cache, which is the same size as in the first UltraSPARC chip, introduced 10 years ago). Moreover, SMT chips already operate on a reduced effective instruction cache size, since the instruction cache is shared among all simultaneous threads. In future processors, the combined effect of larger L2 cache sizes and small (or shared) L1-I caches will make instruction cache stalls the key performance bottleneck.

2.2 Database Workloads on Modern Processors

Prior research [Maynard et al. 1994] has indicated 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 [Ailamaki et al. 1999; Keeton et al. 1998] have narrowed the primary memory-related bottlenecks to L1 instruction and L2 data cache misses. More specifically, Keeton et al. [1998] measured an instruction-related stall component of 41% of the total execution time for Informix running TPC-C on a PentiumPro. When running transactional (TPC-B and TPC-C) and decision-support (TPC-H) benchmarks on top of Oracle on Alpha processors, instruction stalls have accounted for 45% and 30% of the execution time, respectively [Barroso et al. 1998; Stets et al. 2002]. A recent study of DB2 7.2 running TPC-C on Pentium III [Shao et al. 2005] attributed 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 requests reduce the effectiveness of single-query optimizations [Jayasimha and Kumar 1999]. Finally, OLTP instruction streams have strong data dependencies that limit instruction-level parallelism opportunities, and irregular program control flow that undermines built-in pipeline branch prediction mechanisms and increases instruction stall time.

2.3 Techniques to Address L1-I Cache Stalls

In the last decade, research on cache-conscious database systems has primarily addressed data cache performance [Shatdal et al. 1994; Chilimbi et al. 2000; Graefe and Larson 2001; Ailamaki et al. 2001a]. L1-I cache misses, however, and misses occurring when concurrent threads replace each other's working sets [Rosenblum et al. 1995], have received little attention by the database

[1]As of the time this article was written, all major chip manufacturers (Sun, IBM, Intel, AMD) had made available CMP and/or SMT designs (Intel's Pentium 4 implements a two-way SMT design which is marketed as *Hyperthreading* design).

community. Two recent studies [Padmanabhan et al. 2001; Zhou and Ross 2004] proposed 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 [Romer et al. 1997; Ramirez et al. 2001]. In the example of Figure 1 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 and, 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 [Chen et al. 1997]. Call graph prefetching [Annavaram et al. 2003] 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 [Annavaram et al. 2003]).

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[2]). 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 basic 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

[2]Transaction Processing Performance Council. Go online to `http://www.tpc.org`.

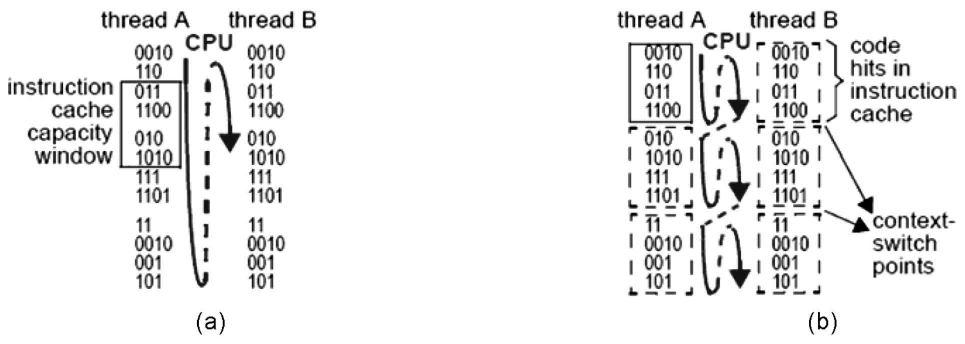


Fig. 3. (a) As the instruction cache cannot fit the entire code, when the CPU switches (dotted line) to thread B, this incurs the same number of misses. (b) By "breaking" the code into three pieces that fit in the cache, and switching back and forth between the two threads, thread B finds all instructions in the cache.

DBMS functions and can easily overwhelm the L1-I cache of modern processors. Experimenting with the Shore database storage manager [Carey et al. 1994] on a CPU with a 64-kB 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 the action that all N transactions are about to perform (except for minor, key-value processing), we follow the code execution for one transaction and find 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 (which are all conflict misses, since each code fragment's footprint is smaller than the L1-I cache).

Figures 3(a) and 3(b) 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. Note that, throughout this section, we consider only function calls made to the DBMS by a transaction, and not the user code surrounding those calls. Further, we assume that all threads are already synchronized and are about to execute the same function call (such as retrieving or inserting a tuple). This way, in this section, we can focus on evaluating the basic STEPS implementation. Later, in Section 4, we remove the above-mentioned assumptions, and describe the full implementation, capable of handling arbitrary transactional workloads (including arbitrary user code) in full-system operation.

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.

3.2 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 typically less costly than process switching. Most commercial DBMS involve a light-weight mechanism to pass on CPU control (Shore uses user-level threads). The code of typical context-switching mechanisms, however, occupies a significant portion of the L1-I cache and takes hundreds of processor cycles to run. Shore's context-switch, for instance, occupies half of Pentium III's 16-kB L1-I cache.

To minimize the overhead of context-switching among STEPS threads, we employ a well-known design guideline: we make the common case fast. The common case here is switching between transactions that are already synchronized and ready to execute the same DBMS function call. Our fast context-switch mechanism executes only the core code and updates only CPU state, ignoring thread-specific software structures (such as the ready-thread queue) until they must be updated. We describe when we update those structures, along with how transactions are synchronized in the first place, in Section 4.

The absolute 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. After adding the code to handle scheduling between different STEPS threads, we derive a context-switching code with a 76-bytes footprint. 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.

3.3 Finding Context-Switching Points in Shore

So far, we have described a fast context-switching function that can be used among threads in a controlled setting (threads are already synchronized and ready to execute the same DBMS function call). The next step is to find appropriate places in the DBMS source code to insert a call to the context-switch function (CTX). These places would correspond to the points where the instruction cache fills up during the execution of a single thread (see also Figure 3(b)). To find these points in the source code, we perform a manual, test-and-try search on all transactional operations in Shore (begin, commit, index fetch, scan, update, insert, and delete). Although our manual search was not particularly lengthy (a few days for one person), we describe later (Section 5) a tool that automates this procedure and that can work with arbitrary code bases.

Given a specific transactional DBMS function call (for example, index-fetch), the manual search proceeds as follows. We create a simple, synthetic database and create two threads that only perform an index-fetch. We pick a random

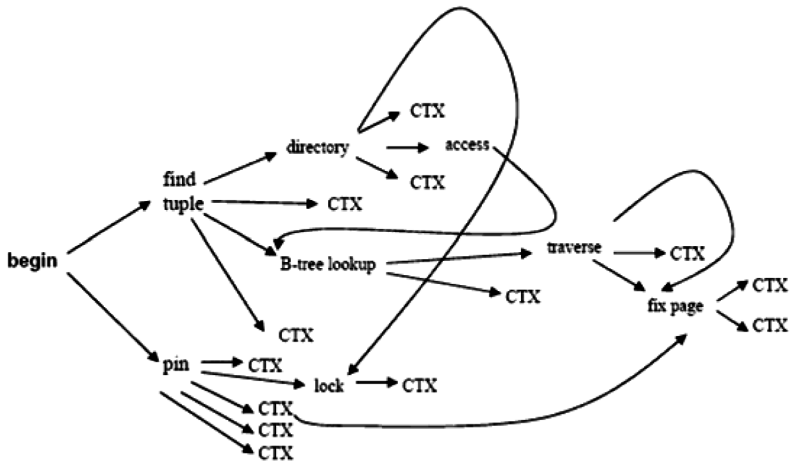


Fig. 4. Context-switch (CTX) calls placed in Shore's source code for index-fetch. Arrows represent code paths, to be followed from left to right and from top to bottom until a CTX is encountered. Note that the actual number of context-switches can be larger than the number of CTX, since a CTX can be placed inside a for loop (i.e., for each level of a B-tree).

point in the source code of index fetch and, using hardware CPU counters[3] (available on most processors), we measure the L1-I cache misses for a single thread executing the code fragment from the beginning of index-fetch up to the point we picked. We then insert in that point a candidate CTX call to the second thread and measure the total number of L1-I misses for executing the code fragment twice (once per thread). If the increase in L1-I misses for the second thread is small ($<5\%$), we pick another point further on in the flow of code and repeat the measurements; otherwise we pick a point earlier in the flow of code. A final CTX point is registered as soon as we detect a knee in the curve of the total number of L1-I cache misses. We treat that CTX point as the beginning of the next code fragment, and measure the L1-I misses from that CTX point up to the next candidate point. We repeat this procedure and insert as many CTX calls as needed to keep the L1-I misses of the second thread (which follows the first in lock-step) low, for the entire index-fetch call. We then continue this search for the rest of the transactional 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. Figure 4 shows the context-switch calls (CTX) placed in index-fetch in Shore, using a machine

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

Table I. Processors Used in Microbenchmarks

CPU	Cache characteristics	
AMD AthlonXP	L1 I + D cache size associativity / block size	64 kB + 64 kB 2-way/64 bytes
	L2 cache size	256 kB
Pentium III	L1 I + D cache size associativity / block size	16 kB + 16 kB 4-way/32 bytes
	L2 cache size	256 kB
Pentium 4	Trace Cache size associativity	12 k uops 8-way
	L1-D cache size associativity / block size	16 kB 8-way/64 bytes
	L2 cache size	1 MB
Simulated IA-32 (SIMFLEX)	L1 I + D cache size associativity	[16, 32, 64 kB] [direct, 2, 4, 8, full]

with 64-kB instruction cache and two-way set associativity. Note that, following this methodology, all threads executing the same transactional operation always remain synchronized while context-switching back-and-forward, since they operate on similarly structured indexes. In Section 4, we show how we handle threads that lose synchronization.

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 use the term *Original* to refer to the original unmodified Shore code and the term *STEPS* to refer to our system built on top of Shore.

3.4 STEPS in Practice: Microbenchmarks

We conduct experiments on the processors shown in Table I. Most experiments (Sections 3.4.1–3.4.3) were run on the AthlonXP, which features a large, 64-kB L1-I cache. The Pentium 4 implements a nonconventional instruction-cache architecture, called *Trace Cache*; we provide background in Section 3.4.4. High-end installations typically run OLTP workloads on server processors (such as the ones shown in Figure 2(b)). 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.

Shore runs under Linux 2.4.20 (2.6 for the Pentium 4 experiments). We used PAPI [Browne et al. 1999] and the perfctr library to access the AthlonXP, Pentium III, and Pentium 4 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 45 kB, as measured by a cache simulator (described in Section 3.4.5). Repeatedly running index-fetch incurred no additional misses in a 45K *fully associative* cache, but it will incur conflict misses in lower-associativity caches,

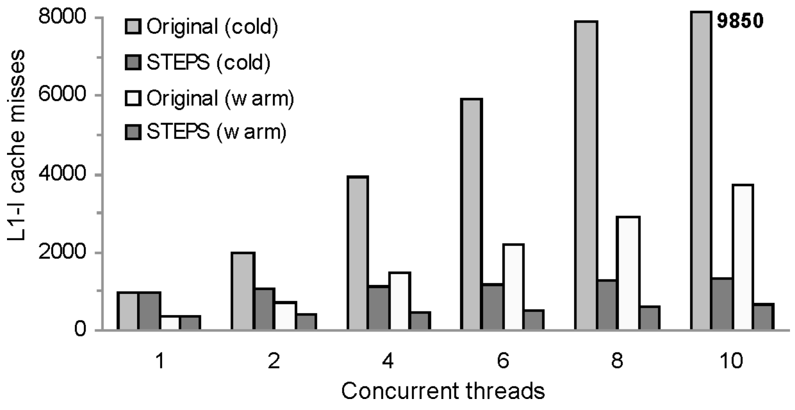


Fig. 5. Proof of concept: STEPS significantly reduced instruction-cache misses as the group size of concurrent threads increased, both with cold and with warm caches.

as explained in Figure 1. We report results averaged over 10 threads, each running index-fetch 100 times.

3.4.1 *Instruction Misses and Thread Group Size.* We measured L1-I cache misses for index-fetch, for various thread group sizes. Both the original system and Shore with STEPS execute the fast CTX call, but STEPS multiplexes thread execution, while the original system executes the threads serially. We first started with a cold cache and flushed it between successive index-fetch calls, and then repeated the experiment starting with a warm cache. Figure 5 shows the results on the AthlonXP.

STEPS incurred only 33 misses for every additional thread, with both a cold and a warm cache. Under the original system, each additional thread added 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 45 kB). 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 warmed up Shore with the same operation, and used the fast CTX call, therefore reporting worst-case lower bounds.

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 that 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 $N \cdot a \cdot m_P$, $0 < a \leq 1$ misses because of fewer capacity misses. In STEPS, all threads except the first incur $\beta \cdot m_P$ misses each, 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:

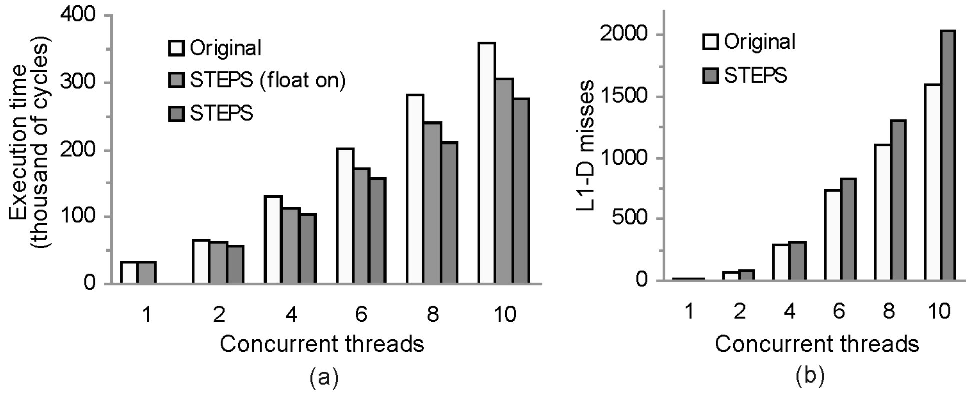


Fig. 6. (a) Execution time for 1 to 10 concurrent threads. STEPS with float on always updates floating point. (b) L1-D cache misses for 1 to 10 concurrent threads.

$a \cdot m_P + (N-1) \cdot \beta \cdot m_P$. When comparing STEPS to the original system, we express the miss reduction percentage as (1 − # misses after /# misses before) · 100%. Therefore, the bounds for computing the L1-I cache miss reduction are

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

where N = group size.

For index-fetch, we measured $a = 0.373$, $\beta = 0.033$, 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 were $a = 0.35$ and $\beta = 0.044$.

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

3.4.2 *Speedup and Level-One Data Cache Misses.* Keeping the same setup as before and providing the original system with a warmed-up cache, we measured execution time in CPU cycles and the number of level-one data (L1-D) cache misses on the AthlonXP. Figure 6(a) shows that STEPS speedup increased with the number of concurrent threads. We plotted both STEPS performance with a CTX function that always updates floating point registers (float on) and with a function that skips updates. The speedup[4] for 10 threads was 31% while for a cold cache it was 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 fine granularity results in a larger collective working set which can overwhelm the L1-D cache (when compared to the original execution sequence). Figure 6(b) shows that STEPS incurred increasingly more L1-D

[4]Throughout the article we compute speedup as SlowerTime/FasterTime. For example, if the original time is 10 s and the improved one is 8 s, the speedup is 1.25 or 25%.

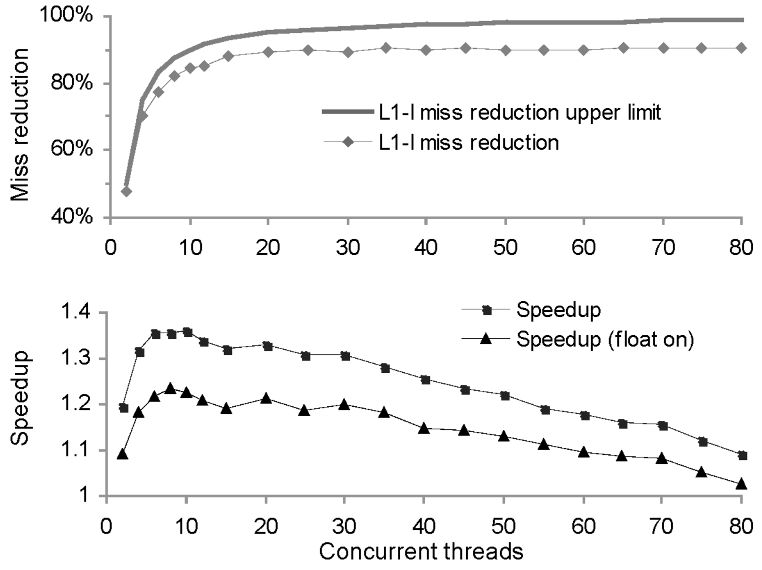


Fig. 7. Lower bounds for speedup using a warm cache for Shore (bottom graph) and percentage of reduction in L1-I cache misses (top graph) of STEPS over Shore, for 2–80 concurrent threads. The top line shows the maximum possible reduction.

cache misses as the thread group size increased. For up to four threads, however, the collective working set had comparable performance to single-thread execution. Fortunately, L1-D cache misses have minimal effect on execution time (as also seen by the speedup achieved). The reason is that L1-D cache misses that hit in the L2 cache can be easily overlapped by out-of-order execution [Ailamaki et al. 1999]. Moreover, in the context of simultaneous multithreading (SMT), it has been shown that, for eight threads executing simultaneously an OLTP workload and sharing the CPU caches, additional L1-D misses can be eliminated [Lo et al. 1998].

On the other hand, there is no real incentive for 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, 256-kB 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. In practice, we have found that statically restricting the group size is sufficient to keep the collective data working set in the L2 cache. Transactional operators are typically characterized by long sequences of instruction streams that touch relatively few memory addresses (when compared to DSS-style tuple-by-tuple processing algorithms) in the time it takes to fill up the L1-I cache with instructions. If we were to apply STEPS to an operator with a particularly large data working set (or to a CPU with a small L2 cache size), then we would also need to implement a dynamic mechanism for restricting the thread group size.

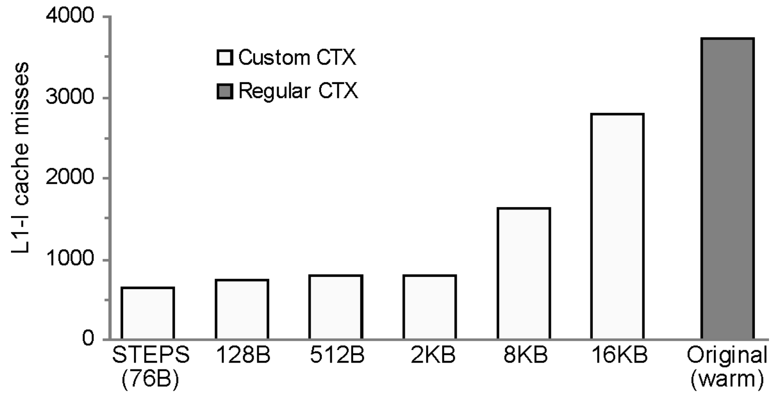


Fig. 8. Number of L1-I cache misses for 10 concurrent threads executing index-fetch, varying the footprint of the context-switch code. The leftmost bar corresponds to STEPS CTX code, the intermediate bars correspond to STEPS CTX padded with consecutive *nops*, and the rightmost bar provides a comparison with the default number of L1-I cache misses with no STEPS, when the cache is warm.

3.4.3 *Increasing the Size of Context-Switching Code.* Next, we examine the effect of the context-switch code size on L1-I cache misses for STEPS when keeping the CTX points in the source code the same. In our implementation, the CTX code size was 76 bytes. For this experiment, we padded the CTX function with a varying number of *nops*[5] to achieve CTX code sizes of up to 16 kB. Figure 8 shows the total number of L1-I cache misses for 10 threads executing index-fetch, for various sizes of the CTX code (128 B to 16 kB). The leftmost bar corresponds to our original STEPS implementation, while the rightmost bar corresponds to the default Shore configuration with no STEPS, under a warmed-up cache. This experiment shows that the CTX code could possibly include more functionality than the bare minimum and still provide a significant reduction in the number of instruction cache misses. In our setup, a CTX function that is 25 times larger than the one we used, would result in almost the same miss reduction (albeit with a lower overall speedup improvement). Having this flexibility in the footprint of the CTX code is important for portability of the STEPS implementation.

3.4.4 *Detailed Behavior on Three Different Processors.* The next experiment examined changes in hardware behavior between STEPS and the original system for index-fetch with 10 threads. We experimented with three processors: the AthlonXP, the Pentium III, and the Pentium 4. The Pentium III features a smaller, 16-kB L1-I and L1-D cache (see also Table I for processor characteristics). Since the CTX points in Shore were chosen when running on the AthlonXP (64-kB L1-I cache), we expected that this version of STEPS on the Pentium III would not be as effective in reducing L1-I cache misses as on the AMD. We refrained from “re-training” STEPS on the Pentium III and the Pentium 4, so

[5]A nop is a special assembly instruction used for padding cache blocks; it provides no other functionality.

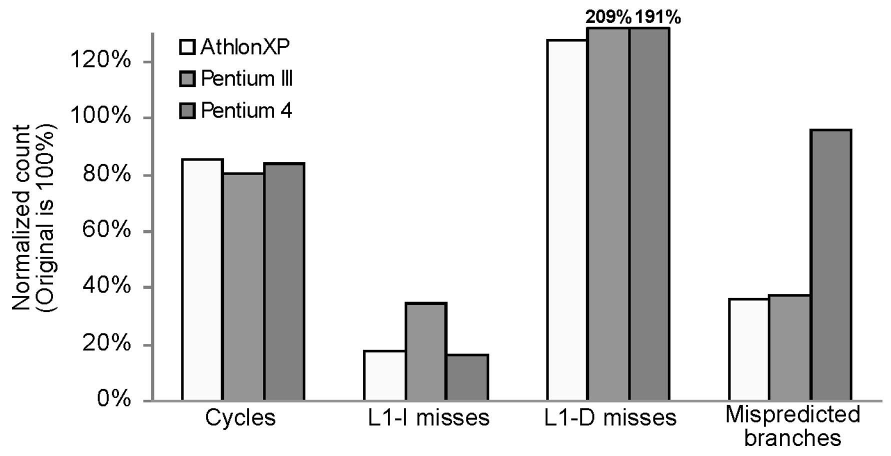


Fig. 9. Relative performance of Shore with and without STEPS, for index-fetch with 10 concurrent threads, on the AthlonXP, the Pentium III, and the Pentium 4.

that we could compare executables with the exact same instruction count (an optimized STEPS implementation for the Pentiums would include more CTX calls).

The Pentium 4 implements a nonconventional instruction cache architecture called *trace cache* [Rotenberg et al. 1996]. A trace cache increases instruction fetch bandwidth by storing traces of instructions that have already been fetched. A trace contains only instructions whose results are actually used, and eliminates instructions following taken branches (since they are not executed). This allows the instruction fetch unit of the processor to fetch several instruction sequences, without having to worry about branches in the execution flow. The trace cache of Pentium 4 stores already decoded microoperations (which are translations of complex x86 instructions), so that the next time an instruction is needed, it does not have to be decoded again.

The results are shown in Figure 9:

—*Execution time and L1-I cache misses.* STEPS was 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 III was slightly higher than the AthlonXP, mainly because the absolute number of misses saved was higher. The Pentium 4 exhibited an instruction-cache miss reduction similar to that of the AthlonXP (both in relative and absolute number of misses), which means that the CTX calls used were as effective as those used by the AthlonXP. The speedup was also similar, though we would expect a higher speed benefit for the Pentium 4 since the L2 takes more cycles to access than in the AthlonXP (due to the higher clock frequency). It turned out, as we discuss below, that Pentium 4 exhibited worse L1-D cache behavior than the AthlonXP, and furthermore, contrary to what happened with the AthlonXP, the Pentium 4 did not reap any benefits from fewer mispredicted branches, since the trace cache already exhibited good branch prediction.

—*Level-one data cache.* STEPS incurred significantly more L1-D cache misses on the Pentiums' small L1-D cache (109% and 91% more misses for the Pentium III and Pentium 4, respectively). However, the CPU coped well by overlapping misses (as exhibited by the 24% speedup).

—*Mispredicted branches.* STEPS reduced mispredicted branches to almost a third on both the AthlonXP and the Pentium III (it eliminated 64% of the original system's mispredicted branches). This is an important result coming from STEPS' ability to provide the CPU with frequently repeating execution patterns. On the other hand, STEPS had no effect on the Pentium 4's branch prediction rate, since the trace cache already exhibited high branch prediction accuracy.

—*Other events* (*not plotted*). L2 cache performance did not have an effect on the specific microbenchmark since almost all data and instructions could be found there. We report L2 cache performance in the next section, when running a full OLTP workload. Also, as expected, STEPS executed slightly more instructions (1.7%) and branches (1.3%) due to the extra context-switch code.

3.4.5 *Varying L1-I Cache Characteristics.* The last microbenchmark varies L1-I cache characteristics using SimFlex [Hardavellas et al. 2004], a Simics-based [Magnusson et al. 2002], 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 modified the L1-I cache characteristics (size, associativity, block size) and ran the 10-thread index-fetch benchmark. The reported L1-I cache misses are 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 10(a), 10(b), and 10(c) show the results for a fixed 64-byte cache block size, varying associativity for 16-, 32-, and 64-kB L1-I caches.

As expected, increasing the associativity reduced instruction conflict misses (except for a slight increase for fully associative 16- and 32-kB caches, due to the LRU replacement policy resulting in more capacity misses). The conflict miss reduction for STEPS was more dramatic in a small cache (16 kB). The reason is that, with a 45-kB 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 two-way 64-kB 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 performed 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 eight-way 32- and 64-kB caches. Experiments with different cache block sizes (not shown here) found that larger blocks further reduced L1-I misses, in agreement with the results in Ranganathan et al. [1998].

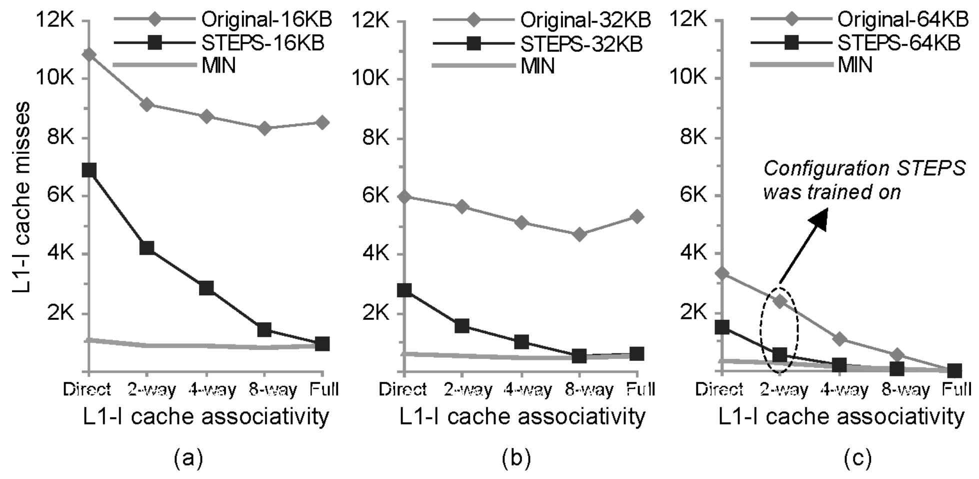


Fig. 10. (a), (b), (c) Simulation results for index fetch with 10 threads. We used a L1-I cache with a 64-byte cache block, varying associativity (direct, two-, four-, eight-way, full) and size (16 kB, 32 kB, 64 kB). STEPS eliminated all capacity misses and achieved up to 89% overall reduction (out of 90% max possible) in L1-I misses (max performance was for the eight-way 32- and 64-kB caches).

4. APPLYING STEPS TO OLTP WORKLOADS

So far we have seen how to efficiently multiplex the execution of concurrent threads running the same transactional DBMS operation when (a) those threads are already synchronized, (b) the threads run uninterrupted, and (c) 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 while maintaining high concurrency for similarly structured DBMS operations, for arbitrary transactions and arrival sequences, 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 (Section 4.2) and the TPC-C results (Section 4.3).

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 and processes/threads which are blocked for any reason). Once a team is formed, 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.

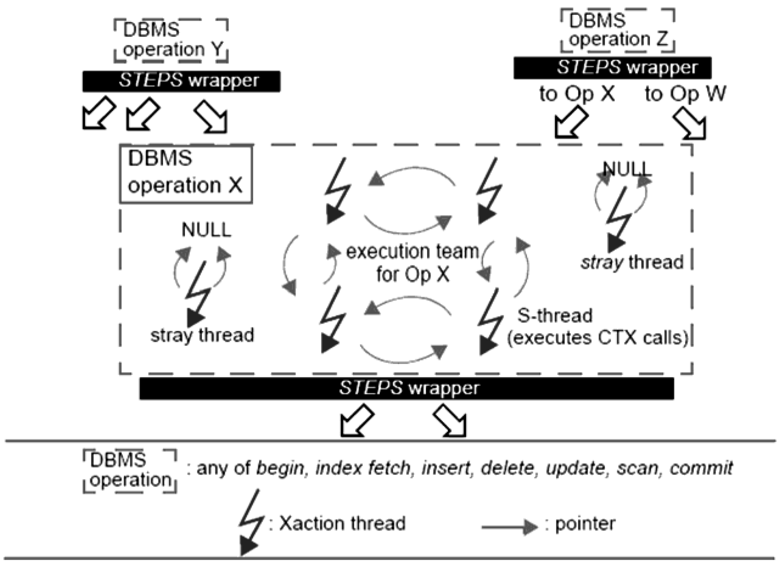


Fig. 11. Additions to the DBMS code: threads are associated with list nodes and form per-operation lists, during the STEPS setup code at the end of each DBMS operation.

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 11). 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.

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 11). 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.[6]

[6]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 in worked well in practice since the system scheduler makes sure that every thread makes progress.

Table II. 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	✓		

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 fashion, the same way as described 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 (Section 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 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 maintain a different list for creating a record and a different one for updating an index.

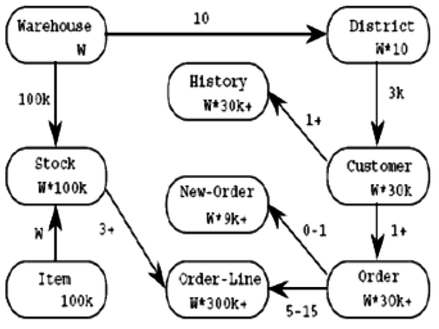
Table II shows all transactional operations along with their degree of cross-transaction overlapped code. *Begin*, *commit*, *scan*, and *update* are independent of the database structure and use a single list each. *Index fetch* code follows different branches depending on the B-tree depth; therefore a separate list per index maximizes code overlap. Last, *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 found that following only the first guideline (declaring lists per index) was sufficient. Small variations in the code path were unavoidable (e.g., utilizing a different attribute set or manipulating different strings) but the main function calls to the DBMS engine were 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 modified the thread package and intercepted 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 bypassed subsequent CTX calls and fell under the authority of the regular scheduler. They remained stray until they joined 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 nongranted 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 nonzero, 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 nonzero.



New Order

```
begin
fetch (D) lock
fetch (W)
fetch (C)
update (D)
for (avg 10)
.....fetch (I)
.....fetch (S) lock
.....update (S)
.....insert (O-L)
insert (O)
insert (N-O)
commit
```

Payment

```
begin
fetch (D) lock
fetch (W) lock
scan (C) 60% prob.
fetch (C) lock
fetch (C) lock, 10% prob.
update (C)
update (D)
update (W)
insert (H)
commit
```

Fig. 12. Database schema for TPC-C benchmark and code outline (in terms of system calls to the DBMS) of the two most frequently executed transactions, New Order and Payment (capital letters correspond to TPC-C tables; fetch and scan are implemented through SQL SELECT statements that retrieve single or multiple records correspondingly.

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 also 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 experimented with TPC-C, the most widely accepted transactional benchmark, which models a wholesale parts supplier operating out of a number of warehouses and their associated sales districts.[7] The benchmark is designed to represent any industry that must manage, sell, or distribute a product or service. It is designed to scale just as the supplier expands and new warehouses are created. The scaling requirement is that each warehouse must supply 10 sales districts, and each district must serve 3000 customers. The database schema along with the scaling requirements (as a function of the number of warehouses W) is shown in Figure 12 (left part).

The database size for one warehouse is 100 MB (10 warehouses correspond to 1 GB and so on). TPC-C involves a mix of five concurrent transactions of different types and complexity. These transactions include entering orders (the New Order transaction), recording payments (Payment), delivering orders, checking the status of orders, and monitoring the level of stock at the warehouses. The first two transactions are the most frequently executed (88% of

[7]Transaction Processing Performance Council. Go online to `http://www.tpc.org`.

Table III. System Configuration

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

any newly submitted transaction), and their code outlines (in terms of calls to the DBMS in our implementation of TPC-C on top of Shore) are shown in Figure 12.

The TPC-C toolkit for Shore is written at Carnegie Mellon. Table III 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/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 avoided overheads when inserts were spread across many transactions). We modified New Order slightly by removing one insert from inside a for-loop (but kept the remaining inserts).

For all the experiments, we warmed up the buffer pool and measured CPU events in full-system operation, including background I/O processes not optimized using STEPS. Measurement periods ranged from 10 s to 1 min, depending on the time needed to complete a prespecified number of transactions. All reported numbers were consistent across different runs, since the aggregation period was large in terms of CPU time. Our primary metric was the number of L1-I cache misses as it was not affected by the AthlonXP's small L2 cache (when compared to server processors shown in Figure 2(b)).

In regard to the STEPS setup, we kept the same CTX calls used in the microbenchmarks but without using floating point optimizations, and without retraining STEPS on TPC-C indexes or tables. Furthermore, we refrained from using STEPS on the TPC-C application code. Our goal was to show that STEPS is workload-independent and to report lower bounds for performance metrics by not using optimized CTX calls. We assigned 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 had no effect since in our configuration the number of runnable threads was low. For larger setups, STEPS can be configured to restrict team sizes, essentially creating multiple independent teams per DBMS operation.

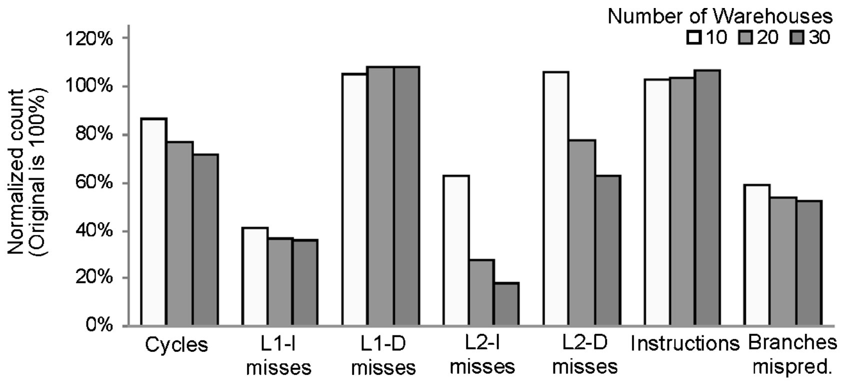


Fig. 13. Transaction mix includes only the Payment transaction, for 10–30 Warehouses (100–300 threads).

Table IV. 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**	

4.3 TPC-C Results

Initially we ran all TPC-C transaction types by themselves, varying the database size (and number of users). Figure 13 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 were 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 outperformed the original system, 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 increased as the database size (and number of users) scaled up. The increase in L1-D cache misses was marginal. STEPS speedup was also fueled by fewer L2-I and L2-D misses as the database size increased. STEPS made better utilization of AMD's small L2 cache as fewer L1-I cache misses also translated into more usable space in L2 for data.

Table IV shows for each configuration (10, 20, and 30 Warehouses running Payment) how many threads on average entered an execution team for a DBMS operation and exited without being stray, along with how many threads were 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

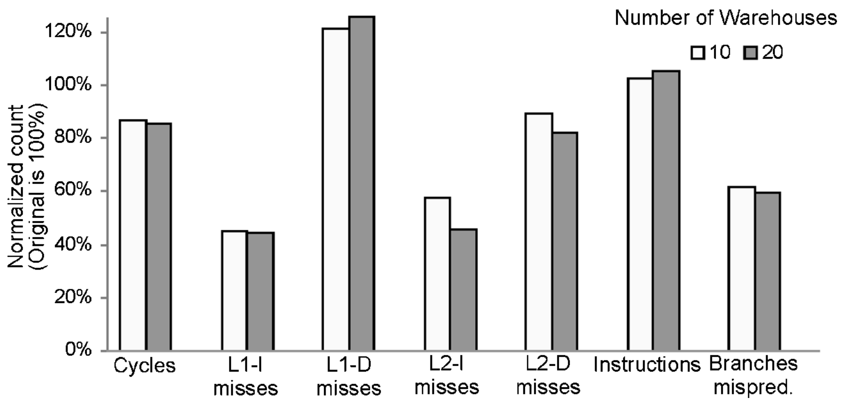


Fig. 14. Transaction mix includes all transactions except the noninteractive Delivery transaction, for 10–20 Warehouses.

(Customer, District, Warehouse, and History). STEPS was able to group on average half of the available threads. Most of the operations yielded a low rate for producing strays, except for *index fetch* on District and Warehouse. In small TPC-C configurations, exclusive locks on those tables restricted concurrency.

Next we ran the standard TPC-C mix, excluding the noninteractive Delivery transaction (TPC-C specifies up to an 80-s queueing delay before executing Delivery). Figure 14 shows that the four-transaction mix followed 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 forced a smaller average team size due to the increased number of unique operations. For the 10-Warehouse configuration, there were 14 ready threads, and on average, 4.3 threads exited from a list without being stray. Still, this means a theoretical bound of a 77% reduction in L1-I cache misses, and STEPS achieved 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.

5. APPLICABILITY TO COMMERCIAL DBMS

STEPS has the following two attractive features that simplify integration in a commercial DBMS: (a) it applies incrementally since it can target specific calls to the DBMS and also can coexist with other workloads which do not require a STEPS runtime, (e.g., decision support applications simply bypass all 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*: programmers develop DBMS code anticipating random context-switches that can occur at any time. Thread-safe code ensures that any placement of CTX calls throughout the code will not break any assumptions.

To apply STEPS to a thread-based DBMS, the programming team first needs to augment the thread package to support fast context-switching. Database software using processes instead of threads may require changes to a larger subset of the underlying OS code. For the rest of this section, we target DBMS software that assigns transactions to threads (and not processes). In general, a STEPS fast CTX call needs to bypass the operating system's scheduler and update only the absolute minimum state needed by a different thread to resume execution. Whenever a thread gives up CPU control through a mechanism different than fast CTX (e.g., disk I/O, unsuccessful lock requests, failure to enter a critical section, or expired time quantum), all state needed before invoking a regular context-switch needs to be updated accordingly. This is the state that the OS scheduler needs to make scheduling decisions. The next phase is to add a STEP wrapper in each major DBMS transactional operation. This thin wrapper will provide the high-level, per-operation transaction synchronization mechanism used in STEPS. The only workload-specific tuning required is the creation of per-index execution teams, which can be done once the database schema is known. The previous two sections described our approach into implementing the above mentioned guidelines in Shore running on Linux. The implementation does not depend on the core DBMS source code, since it only affects the thread package and the entry (or exit) points of a few high-level functions.

The final phase in applying STEPS is to decide how many fast CTX calls to insert in the DBMS code and where exactly to place them. So far, we have identified candidate insertion points by "test-and-try." We performed a manual search by executing DBMS operations on the target hardware, using the CPU performance counters to count instruction-cache misses. In our implementation, we could afford the time to perform a manual search since Shore's code is relatively small (around 60,000 lines of code). Commercial systems, however, may contain tens of millions of code lines. To aid STEPS deployment in large-scale DBMS software, we need a tool that can automatically decide on where to insert CTX calls throughout the source code. Moreover, such a tool should be versatile enough to produce different outputs for different targeted cache architectures. This way, a DBMS binary with STEPS can ship optimized for a specific CPU.

In the remainder of this section, we describe autoSTEPS, our approach toward automating the deployment of STEPS in commercial DBMS. Section 5.1 presents the functionality and usage of the tool, Section 5.2 describes the implementation, while Section 5.3 evaluates the accuracy of autoSTEPS.

5.1 autoSTEPS: A Tool to Generate CTX Insertion Points

We leverage existing open-source software to obtain a trace of all instruction references during the execution of a transactional operation. Valgrind[8]/cachegrind[9] is a cache profiling tool which tracks all memory references (both data and instruction) from a binary executable and passes them through a cache simulator to report statistics. We modify cachegrind to output all memory addresses of instructions executed between two "magic" instructions, placed

[8]Web site: `http://valgrind.org/`.
[9]Web site: `http://valgrind.org/docs/manual/cg_main.html`.

anywhere in the DBMS source code (or user application code). autoSTEPS is a cache-simulator script, written in Python, that takes as input the memory address trace and outputs the source code lines where fast CTX calls should be placed for STEPS-like execution. The tool usage is the following:

—First, the user specifies the DBMS operation for processing (such as index-fetch, insert, update, or any desired function call), by creating a sample client application which contains the operation to be profiled.
—If the targeted platform is the same as the one used to run the tool, no further action is needed (the tool automatically extracts cache parameters and outputs them for verification). Otherwise, the user needs to specify the cache characteristics (L1-I cache size, associativity, and block size) using a switch.
—The user first executes the modified version of valgrind to obtain the trace (i.e., "`>steps_valgrind sample_transaction.exe > trace`"), and then runs autoSTEPS on the collected trace.
—The tool outputs the memory addresses of the code that CTX calls need to be inserted, along with various cache statistics, both for plain and STEPS execution. Note that, depending on the underlying cache architecture, the tool may output different addresses for the CTX calls.

A specific DBMS operation can be profiled for STEPS by compiling a client application that issues sample transactions to a set of tables in the database. If the targeted operation is a function call exported by the DBMS to the client application, then the user inserts a "magic" instruction right before and right after the function call to the DBMS, and recompiles the client application. If the function call to be profiled is an internal one, then the magic instruction should be inserted at the DBMS source code. The magic instruction is a unique sequence of native assembly instructions with no effect, which differs on each platform and comes with the tool documentation; its purpose is to instruct the tool to start and stop processing the binary.

To translate the memory addresses to source code line numbers, autoSTEPS invokes the system debugger in batch mode and outputs source file names and line numbers. Once the lines in the source code are known, the user simply inserts CTX calls in place (the code of the CTX call is always the same). Then the user recompiles and runs the transactional application. Note that this procedure does not guarantee a deadlock-free execution since it may have introduced potential races. In our experiments with Shore, this was not the case, but it may occur in other systems. The autoSTEPS tool is a tool that aids in the process of instrumenting code, not a complete binary solution. We envision that a commercial application would also include two more tools. The first is a binary modification tool, similar to the one described in Srivastava and Eustace [1994], used to insert the CTX calls directly to the DBMS binary with no need for recompiling. The second tool is a race-detection binary tool, similar to the one described in Savage et al. [1997], used to pinpoint badly placed CTX calls which may cause races or force S-threads to go astray. Since similar tools already exist, it is out of the scope of this work to reimplement such functionality. Next we describe the implementation of autoSTEPS.

5.2 autoSTEPS Algorithm

To profile code for STEPS, we only need to examine L1-I cache traffic. We reimplement the cache simulator of cachegrind, to compare with STEPS-like execution. To find points in the code to place fast CTX calls, we need to consider cache misses only for *nonleading*[10] threads in an execution team. A regular cache simulator will count all misses for the single executing thread. Under STEPS, these are the default, compulsory misses that will always be caused by the leading thread. Since STEPS performance is governed by the misses caused by nonleading threads, we need to track which cache blocks are *evicted* during execution of a code segment by the leading thread. Those evicted cache blocks will need to be reloaded when nonleading threads execute the same code segment (immediately after the next CTX call). This way, we can decide on where to place CTX calls and also compute overall misses for all threads in the execution team, by processing a trace of only one thread executing the entire transactional operation.

Our algorithm works on top of the regular L1-I cache simulation. Starting from the beginning of the trace, the autoSTEPS simulator marks each cache block accessed with the number of the current code segment. These are the cache blocks loaded by the leading thread and, initially, the segment number is 0. The algorithm does not take into consideration regular misses. However, whenever a cache block with the same segment number as the currently executed segment is evicted, we count it as a *STEPS miss*. A STEPS miss is a miss that would have been caused by a nonleading thread. When the number of STEPS misses reaches a threshold, we mark the current memory address as the place where a CTX call will be inserted, increase the code segment number, reset the number of STEPS misses, and continue processing the trace.

To better match CTX call selection with code behavior, instead of using a fixed threshold for STEPS misses, we place CTX calls close to the beginning of a “knee” in the order that misses occur. Such a knee appears when (a) STEPS misses occur close to each other in terms of executed instructions (the number of in-between instructions is the width of the knee), and (b) when the number of consecutive misses that are spaced out by fewer instructions than the knee width reaches a predefined threshold (the “height” of the knee). Whenever two consecutive STEPS misses are separated by more than the knee width, we reset the knee height counter, up to a maximum of twice the knee height for STEPS misses between two consecutive CTX calls. The input to autoSTEPS is the width of the knee (as a number of instructions) and the height of the knee (as a number of STEPS misses). Upon detection of a knee, the CTX call is placed before the instruction that marks the beginning of the knee. For the

[10]A *leading* thread in an execution team is the first thread to execute (and therefore load in the cache) a new code segment, enclosed by two consecutive CTX calls. *Nonleading* threads are the rest $n - 1$ threads in the execution team that will execute the same code segment and will incur significantly fewer misses.

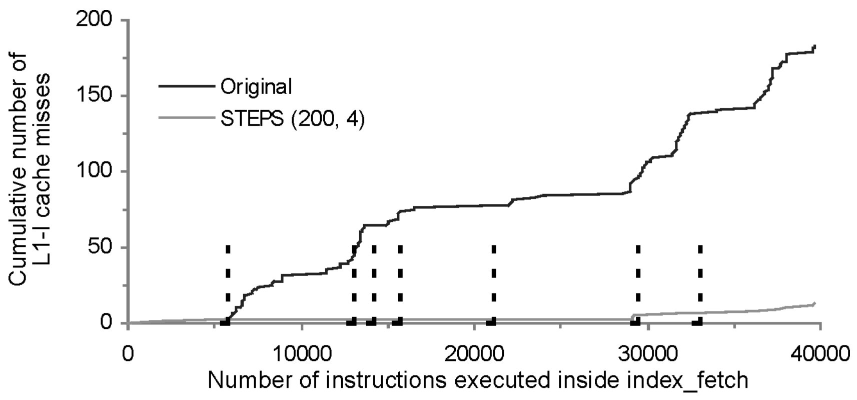


Fig. 15. Cumulative number of instruction cache misses for single-thread execution of index_fetch for all executed instructions. *Original* corresponds to warm-cache behavior for any thread. The line for *STEPS simulation (200,4)* shows misses for any nonleading STEPS thread. The vertical lines show the points where a fast CTX took place, when autoSTEPS is configured for a knee of 200 instructions width and height 4.

code segment following the newly placed CTX call, we can afford more evictions before one results into a STEPS miss (since it is a new segment for all team threads), effectively "absorbing" the knee in the misses altogether.

To test autoSTEPS, we collected a trace of instructions executed during index-fetch, using the same setup as described in Section 3. We ran autoSTEPS with a knee of width 200 instructions and height 4 misses and compared the number of misses for a single thread (nonleading thread when in STEPS mode) for the original code and STEPS execution. Figure 15 shows the cumulative number of misses as those added up for every executed instruction (the number of executed instructions at any time is on the x-axis). The top line corresponds to the L1-I cache misses of the original code, when the cache was warmed up with the exact same operation, as computed by the cache simulator of cachegrind. The bottom line corresponds to STEPS misses (misses caused by nonleading threads), while the vertical lines show where a CTX call took place. In this run, autoSTEPS output three insertion points for CTX code, which resulted in a total of seven CTX calls. As Figure 15 shows, the resulting configuration kept the overall number of misses very low, with only three CTX insertion points in the source code. Note that, during manual tuning, a total of 16 CTX insertion points was used.

To examine the effect of varying the knee input parameters to autoSTEPS, Figure 16 plots the number of L1-I cache misses (y-axis) against the number of CTX calls executed (x-axis) for various inputs. The observed trend is an expected one: as the height of the knee was reduced, autoSTEPS resulted in more recommendations for CTX calls, which brought the overall number of misses lower. For the same height, a wider knee essentially allowed more flexibility in defining a knee, absorbing more misses. Note that these are trends that do not always hold, since autoSTEPS does not try to minimize the actual number of CTX calls executed. A CTX insertion point is picked according to the miss behavior up to that point, and not according to how many total calls it

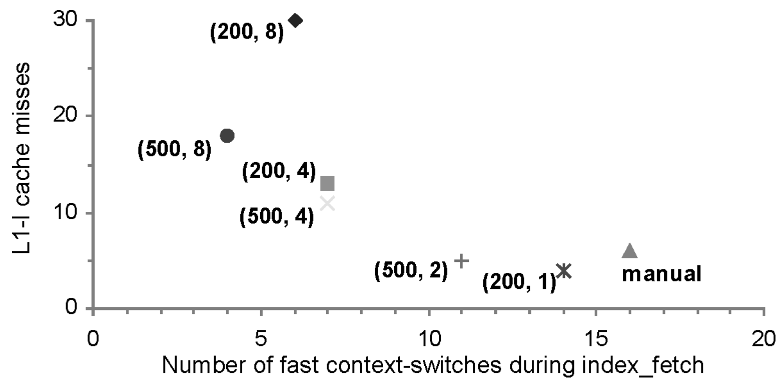


Fig. 16. Number of L1-I cache misses and total number of context-switches executed for nonleading STEPS threads, for different runs of autoSTEPS using different input parameters. The *manual* point is produced by using the same CTX calls as in the manually tuned system.

will generate. Figure 16 also shows the performance of the manually configured STEPS system used in the previous sections. The number of CTX calls in the manually tuned system exceeded those resulting from autoSTEPS recommendations. Note that the cache simulation reported fewer misses for both the original and STEPS execution when compared to the simulation results given in Section 3.4.5, because of the simplicity of the simulator (e.g., it does not simulate prefetching and assumes a perfect LRU replacement policy) and because we are assuming a perfectly warmed-up cache with no pollution from context-switching code.

5.3 Evaluation

To evaluate the effectiveness of the recommendations produced by autoSTEPS, we picked the same input parameters as in Figure 15 (knee width 200 and height 4). autoSTEPS output the line number and source file of three places in Shore's code to insert fast CTX calls. Since the recommendations from Figure 16 refer to arbitrary source lines, and those lines can be inside header files or shared library code, it is not always straightforward to manually insert a CTX call without a binary modification tool, or without adjusting the placement of the call. For the configuration used in this section (200, 4), we were able to recompile the code by simply inserting a CTX call in the output lines of the autoSTEPS tool, but as the number of CTX calls increased for other configurations it became necessary to inspect the source code and make appropriate placement adjustments.

After inserting the CTX calls and recompiling the code, we ran the index-fetch microbenchmark with 10 threads on AthlonXP. We compared three systems: the original Shore, Shore with STEPS using the manual search for CTX insertion points (which resulted in a total of 16 insertions), and Shore with STEPS using the three insertion points recommended by autoSTEPS. We expected that the specific autoSTEPS configuration would not outperform the manual one in reducing the number of misses, as shown in the simulation results of Figure 16, but the reduced number of CTX calls would give a relative speedup benefit to

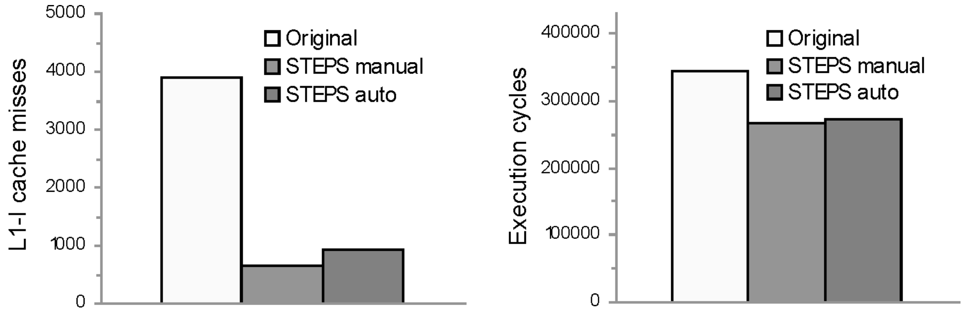


Fig. 17. Performance comparison of Original, STEPS with manually placed CTX calls, and STEPS with an automatically produced configuration. We used autoSTEPS(200,4) to derive only three CTX calls (the manual configuration has 16 CTX calls).

the autoSTEPS configuration. The results are shown in Figure 17. The left part of the figure shows the total number of L1-I cache misses for all 10 threads executing index-fetch, for the three systems. As expected, the autoSTEPS configuration significantly outperformed the original system and was slightly worse than the manual configuration. The right part of Figure 17 shows the total execution cycles. The autoSTEPS configuration performed almost as well as the manual configuration since it executed fewer context-switches.

6. CONCLUSION

This article described STEPS, a transaction coordinating mechanism that addresses the instruction-cache bottleneck in OLTP workloads. As recent studies have shown, instruction-cache misses in transaction processing account for up to 40% of the execution time. Although compiling techniques and recently proposed architectural features can partially alleviate the problem, the database software design itself holds the key for eliminating cache misses by targeting directly the root of the problem. While database researchers have demonstrated the effectiveness of cache-conscious algorithms and data structures on data cache misses, instruction cache performance in transaction processing has yet to be addressed from within the software. The size of the code involved in transaction processing and the unpredictable nature of transaction execution make a software approach to eliminate instruction-cache misses a challenging one.

STEPS is a mechanism that can be applied with few code changes to any database software and shape the thread execution sequence to improve temporal locality in the instruction cache. To achieve this, STEPS first forms teams of threads executing the same system component. It then multiplexes thread execution within a team, at such fine granularity that it enables reuse of instructions in the cache across threads. To the best of our knowledge, STEPS is the first software approach to provide explicit thread scheduling for improving instruction cache performance. STEPS is orthogonal to compiler techniques and its benefits are always additional to any binary-optimized configuration. This article shows 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.

We expect the ideas behind STEPS to become even more relevant and applicable to the upcoming generation of multicore and multithreaded chips. Researchers tend to agree that server software will require fundamental changes to take advantage of the abundant on-chip processing elements in the most efficient way. With STEPS, we show how a small, well-targeted set of changes can affect the entire code base of a DBMS and significantly improve instruction-cache performance in transaction processing workloads. In a multicore environment, an additional layer of core-aware scheduling will be needed to route similarly structured operations to the same core. For multithreaded (SMT) cores, we could further increase performance by mapping team threads to hardware threads. In that case, we would also need to replace the fast context-switch calls with simple synchronization barriers to make sure that all threads remain synchronized. Another possible extension, on the hardware front, would be to implement autoSTEPS on hardware and let the hardware make calls to the fast context-switch function. As instruction caches do not grow in size, we expect related research efforts to remain active in the near future.

ACKNOWLEDGMENTS

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.

REFERENCES

AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND SKOUNAKIS, M. 2001a. Weaving relations for cache performance. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases.* Morgan Kaufmann, San Francisco, CA, 169–180.

AILAMAKI, A., DEWITT, D. J., AND HILL, M. D. 2001b. Walking four machines by the shore. In *Proceedings of the Fourth Workshop on Computer Architecture Evaluation Using Commercial Workloads* (CAECW).

AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND WOOD, D. A. 1999. DBMSs on a modern processor: Where does time go? In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases.* Morgan Kaufmann, San Francisco, CA, 266–277.

ANNAVARAM, M., PATEL, J. M., AND DAVIDSON, E. S. 2003. Call graph prefetching for database applications. *ACM Trans. Comput. Syst. 21*, 4, 412–444.

BARROSO, L. A., GHARACHORLOO, K., AND BUGNION, E. 1998. Memory system characterization of commercial workloads. In *ISCA '98: Proceedings of the 25th Annual International Symposium on Computer Architecture.* IEEE Computer Society Press, Los Alamitos, CA, 3–14.

BROWNE, S., DEANE, C., HO, G., AND MUCCI, P. 1999. PAPI: A portable interface to hardware performance counters. In *Proceedings of Department of Defense HPCMP Users Group Conference.*

CAREY, M. J., DEWITT, D. J., FRANKLIN, M. J., HALL, N. E., MCAULIFFE, M. L., NAUGHTON, J. F., SCHUH, D. T., SOLOMON, M. H., TAN, C. K., TSATALOS, O. G., WHITE, S. J., AND ZWILLING, M. J. 1994. Shoring up persistent applications. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data.* ACM Press, New York, NY, 383–394.

CHEN, I.-C. K., LEE, C.-C., AND MUDGE, T. N. 1997. Instruction prefetching using branch prediction information. In *ICCD '97: Proceedings of the 1997 International Conference on Computer Design* (ICCD '97). IEEE Computer Society Press, Los Alamitos, CA, 593.

CHILIMBI, T. M., HILL, M. D., AND LARUS, J. R. 2000. Making pointer-based data structures cache conscious. *Computer 33*, 12, 67–74.

Graefe, G. and Larson, P. 2001. B-tree indexes and CPU caches. In *Proceedings of the 17th International Conference on Data Engineering.* IEEE Computer Society Press, Los Alamitos, CA, 349–358.

Hardavellas, N., Somogyi, S., Wenisch, T. F., Wunderlich, R. E., Chen, S., Kim, J., Falsafi, B., Hoe, J. C., and Nowatzyk, A. G. 2004. Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *SIGMETRICS Perform. Eval. Rev. 31*, 4, 31–34.

Harizopoulos, S. and Ailamaki, A. 2004. STEPS towards cache-resident transaction processing. In *VLDB '04: Proceedings of the 30th International Conference on Very Large Data Bases.* Morgan Kaufmann, San Francisco, CA, 660–671.

Hennessy, J. L. and Patterson, D. A. 1996. *Computer Architecture: A Quantitative Approach* (2nd ed.). Morgan Kaufmann, San Francisco, CA.

Jayasimha, J. and Kumar, A. 1999. Thread-based cache analysis of a modified TPC-C workload. In *Proceedings of the Second Workshop on Computer Architecture Evaluation Using Commercial Workloads.*

Keeton, K., Patterson, D. A., He, Y. Q., Raphael, R. C., and Baker, W. E. 1998. Performance characterization of a quad pentium pro smp using oltp workloads. In *ISCA '98: Proceedings of the 25th Annual International Symposium on Computer Architecture.* IEEE Computer Society Press, Los Alamitos, CA, 15–26.

Lo, J. L., Barroso, L. A., Eggers, S. J., Gharachorloo, K., Levy, H. M., and Parekh, S. S. 1998. An analysis of database workload performance on simultaneous multithreaded processors. In *ISCA '98: Proceedings of the 25th Annual International Symposium on Computer Architecture.* IEEE Computer Society Press, Los Alamitos, CA, 39–50.

Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hllberg, G., Hgberg, J., Larsson, F., Moestedt, A., and Werner, B. 2002. Simics: A full system simulation platform. *Computer 35*, 2, 50–58.

Maynard, A. M. G., Donnelly, C. M., and Olszewski, B. R. 1994. Contrasting characteristics and cache performance of technical and multiuser commercial workloads. In *ASPLOS-VI: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM Press, New York, NY, 145–156.

Padmanabhan, S., Malkemus, T., Agarwal, R. C., and Jhingran, A. 2001. Block oriented processing of relational database operations in modern computer architectures. In *Proceedings of the 17th International Conference on Data Engineering.* IEEE Computer Society Press, Los Alamitos, CA, 567–574.

Ramirez, A., Barroso, L. A., Gharachorloo, K., Cohn, R., Larriba-Pey, J., Lowney, P. G., and Valero, M. 2001. Code layout optimizations for transaction processing workloads. In *ISCA '01: Proceedings of the 28th Annual International Symposium on Computer Architecture.* ACM Press, New York, NY, 155–164.

Ranganathan, P., Gharachorloo, K., Adve, S. V., and Barroso, L. A. 1998. Performance of database workloads on shared-memory systems with out-of-order processors. In *ASPLOS-VIII: Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM Press, New York, NY, 307–318.

Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., Levy, H., Bershad, B., and Chen, B. 1997. Instrumentation and optimization of win32/intel executables using etch. In *Proceedings of the USENIX Windows NT Workshop.*

Rosenblum, M., Bugnion, E., Herrod, S. A., Witchel, E., and Gupta, A. 1995. The impact of architectural trends on operating system performance. In *SOSP '95: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles.* ACM Press, New York, NY, 285–298.

Rotenberg, E., Bennett, S., and Smith., J. E. 1996. Trace cache: A low latency approach to high bandwidth instruction fetching. In *MICRO 29: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture.* IEEE Computer Society Press, Los Alamitos, CA, 24–35.

Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15*, 4, 391–411.

Shao, M., Ailamaki, A., and Falsafi, B. 2005. DBmbench: Fast and accurate database workload representation on modern microarchitecture. In *Proceedings of the 15th IBM Center for Advanced Studies Conference* (Oct).

Shatdal, A., Kant, C., and Naughton, J. F. 1994. Cache conscious algorithms for relational query processing. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases.* Morgan Kaufmann, San Francisco, CA, 510–521.

Srivastava, A. and Eustace, A. 1994. Atom: A system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation.* ACM Press, New York, NY, 196–205.

Stets, R., Gharachorloo, K., and Barroso, L. A. 2002. A detailed comparison of two transaction processing workloads. In *WWC-5: IEEE 5th Annual Workshop on Workload Characterization.*

Zhou, J. and Ross, K. A. 2004. Buffering database operations for enhanced instruction cache performance. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data.* ACM Press, New York, NY, 191–202.

Received December 2005; revised May 2006; accepted May 2006