Optimistic Intra-Transaction Parallelism on Chip Multiprocessors

Christopher B. Colohan*
Anastassia Ailamaki* **J. Gregory Steffan**†
Todd C. Mowry*

March 2005
CMU-CS-05-118

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
†Department of Electrical & Computer Engineering, University of Toronto, Toronto, Ontario, Canada

This research is supported by grants from the National Science Foundation and from IBM. Anastassia Ailamaki is partially supported by a Sloan Fellowship.

Keywords: thread level speculation, TLS, database systems, chip multiprocessors, OLTP, TPC-C

Abstract

With the advent of chip multiprocessors, exploiting intra-transaction parallelism is an attractive way of improving transaction performance. However, exploiting intra-transaction parallelism in existing database systems is difficult, for two reasons: first, significant changes are required to avoid races or conflicts within the DBMS, and second, adding threads to transactions requires a high level of sophistication from transaction programmers. In this paper we show how dividing a transaction into *speculative threads* solves both problems—it minimizes the changes required to the DBMS, and the details of parallelization are hidden from the transaction programmer. Our technique requires a limited number of small, localized changes to a subset of the low-level data structures in the DBMS. Through this method of parallelizing transactions we can dramatically improve performance: on a simulated 4-processor chip-multiprocessor, we improve the response time by more than a factor of two when running an OLTP workload.

1 Introduction

We are in the midst of a revolution in microprocessor design: computer systems from all of the major manufacturers that feature chip multiprocessors (CMPs) and simultaneous multithreading (SMT) are entering the marketplace. Examples include Intel's "Smithfield" (dual-core Pentium IV's with 2-way SMT), IBM's Power 5 (combinable, dual-core, 2-way SMT processors), AMD's Opteron (dual-core), and Sun Microsystems's Niagara (an 8-processor CMP). How can database systems exploit this increasing abundance of hardware-supported threads? Currently, for OLTP workloads, threads are primarily used to increase transaction *throughput*; ideally, we could also use these parallel resources to decrease transaction *latency*. Although most commercial database systems do exploit *intra-query parallelism* within a transaction, this form of parallelism is only useful for long running queries, while OLTP workloads tend to issue multiple short queries. To the best of our knowledge, commercial database systems do not exploit *intra-transaction parallelism* [2, 11, 20], and for good reason.

Parallelizing a transaction is difficult. First, the DBMS must be modified to support multiple threads per transaction. Latches must be added to data structures which are shared between threads in the transaction. These latches add complexity and hinder performance. Second, the transaction must be divided into parallel threads. Consider the New Order transaction, which is the prevalent transaction in TPC-C [6] (Figure 1). We can parallelize the main loop (which represents 80% of the execution time), such that each loop iteration runs as a thread. The transaction programmer must understand when these threads may interfere with each other, and add inter-thread locks to avoid problems; e.g., the thread should use inter-thread locks to ensure that only one thread updates the quantity of an item in the `stock` table at a time. Finally, the transaction programmer must test the new transaction to ensure that the resulting parallel execution is correct and ensure that no new deadlock conditions or subtle race conditions were introduced, and then repeat the entire process until satisfactory performance is achieved.

1.1 Incremental Parallelization with Thread-Level Speculation

Fortunately, computer architecture researchers have anticipated this problem and have developed hardware support for chip multiprocessors which makes parallelization easier, called *Thread Level Speculation* (TLS) [17, 18, 7]. Support for TLS is simple and elegant, has been implemented in Sun's MAJC [18] and Stanford's Hydra [7] CMPs, and is a strong candidate technology for future high-end CMPs. In a nutshell, TLS allows a program to be divided arbitrarily into speculative threads (or *epochs*) which are executed in parallel. The TLS mechanism ensures that the parallel execution is *identical* to the original sequential execution. It preserves sequential semantics by tracking data dependences between epochs and restarting threads when their execution diverges from the original sequential execution (TLS is described in more detail in Section 2). In essence, dividing a program into epochs improves performance and does not affect correctness.

Our goal is to parallelize important OLTP transactions, to achieve high performance with

```
begin_transaction {
  Read customer info [customer, warehouse]
  Read & increment order # [district]
  Create new order [orders, neworder]
  for(each item in order){
    Get item info [item]
    if(invalid item)
      abort_transaction
    Read item quantity from stock [stock]
    Decrement item quantity
    Record new item quantity [stock]
    Compute price
    Record order info [order_line]
  }
} end_transaction
```

80% of transaction execution time

Figure 1: The New Order transaction. In brackets are the database tables touched by each operation.

low implementation overhead. In this study we parallelize the main loop of the New Order transaction running on the BerkeleyDB DBMS [13]. We execute that transaction on a TLS system which provides profile information identifying the performance bottleneck; we then perform localized optimizations on the DBMS code to remove that bottleneck. Removal of each bottleneck exposes the next bottleneck, so we repeat the process. This paper describes the bottlenecks we encountered (Section 4), and provides general techniques for eliminating them (Section 3).

While the results in this paper are gathered with the New Order transaction running on BerkeleyDB, our techniques can be generalized in two important ways. First, the changes we made were to DBMS data structures and functions which are shared by all transactions, hence the optimizations we describe can be applied to any transaction. Second, we change fundamental primitives used by all database systems (such as latches and locks), hence our techniques are not specific to BerkeleyDB and can be applied to other database systems. Applying our techniques required changing less than 1200 lines out of 180,000 lines of code in the DBMS, and took a graduate student about one month of work. As a result we have doubled the performance of the New Order transaction.

1.2 Related Work

Traditionally, high-performance database systems have targeted inter-transaction parallelism, or intra-operation parallelism, while this paper introduces new techniques for exploiting intra-transaction parallelism. Previous work on intra-transaction parallelism has focused on hand-parallelized transactions [9, 15]—requiring great effort from the programmer and significant changes to the DBMS. In this paper we show how to exploit intra-transaction parallelism with very little effort by the transaction programmer, and minimal changes to the DBMS.

This paper draws upon Kung's optimistic concurrency control work [10] in two ways. First, the execution of *epochs* in thread level speculation is very similar to the execution of optimistic transactions: epochs optimistically assume that they will not conflict with other epochs, epochs compare all of their reads and writes to earlier epochs to ensure a serializable execution, and epochs commit when they succeed or abort and restart when speculation fails. Second, we optimistically omit lock and latch acquires in epochs, and let the TLS mechanism resolve conflicts. Our technique for optimistically omitting locks is also similar to transactional memory [8]. Where the approach to lock elision in this paper diverges from both transactional memory and optimistic concurrency control is that we allow speculative transactions to interact with non-speculative transactions with *no* changes to the non-speculative transactions.

The idea of using speculative execution to simplify manual parallelization was first proposed by Prabhu and Olukotun for parallelizing SPEC benchmarks on the Hydra chip multiprocessor [14, 7]. The base Hydra multiprocessor uses a design similar to the shared cache CPU design used in the evaluation portion of this paper.

1.3 Contributions

This paper makes the following contributions: (i) it solves the problem of parallelizing the central loop of a transaction, which reduces transaction latency and hence decreases contention for resources used by the transaction; (ii) it provides a methodology for eliminating the data dependences which limit parallel performance, describing three specific techniques for eliminating these dependences and examples of their application; (iii) it demonstrates the application of these techniques by incrementally parallelizing a transaction running on a real DBMS, reducing transaction latency by more than a factor of two on a four-CPU machine.

In the next section we explain how the TLS mechanism enables incremental parallelization of transactions. Section 3 describes the three techniques we use to eliminate data dependences between epochs. Section 4 shows how these three techniques are applied by explaining the performance optimizations we performed on the DBMS. In Section 5 we measure the performance gain of each optimization, and we conclude in Section 6.

2 Applying TLS to Transaction Execution

When extracting intra-transaction parallelism, thread level speculation (TLS) allows the programmer to parallelize aggressively without worrying about correctness. Consider a transaction which updates several rows in a database: if the rows are indeed independent, then these updates could be performed in parallel. However, since the row IDs are typically not known in advance, the updates must instead be performed sequentially to preserve potential read and write dependences between different updates to the same row. With TLS support,

(a) Sequential execution.

(b) Parallel execution with TLS.

Figure 2: How TLS ensures that all reads and writes occur in the original sequential order.

the updates could be performed aggressively in parallel, limited only by the actual run-time dependences between rows. The following describes the basic functionality of TLS, including both software and hardware support.

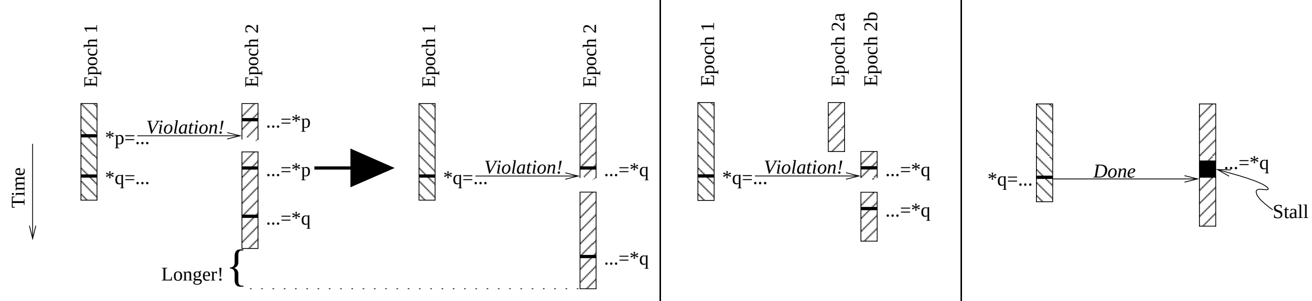
2.1 Software Support for TLS

Under TLS, sequential code (Figure 2(a)) is divided into *epochs*, which are executed in parallel by the system (Figure 2(b)). The system is aware of the original sequential order of the epochs, and also observes every read and write to memory that the epoch performs (i.e. the reads and writes through `p` and `q`).

The system observes whether epoch 1 ever writes to a memory location which has already been read by epoch 2—if so, then epoch 2 has *violated* sequential semantics, and is rewound and re-executed with the correct value. For example, in Figure 2(b) we see that epoch 2 read `p` before epoch 1 wrote to `p`, so we restart epoch 2. On the second execution epoch 2 reads the new value. Note that the read of `q` does not cause a violation, since it executes *after* the write to `q`, and thus reads the correct value. By observing all loads and stores, and restarting an epoch whenever it consumes an incorrect value, the TLS mechanism ensures that the parallel execution is identical to the original sequential execution.

The execution of epochs is similar to the execution of transactions in a system with optimistic concurrency control [10]: an epoch either commits or is violated (aborts), and if there are no dependences between epochs then their concurrent execution will be successful. The difference is that epochs are much smaller than transactions (we demonstrate the use of between 5 and 150 epochs per transaction in this paper), and hardware support makes the cost of violating and restarting an epoch much lower than the cost of aborting and restarting a transaction. In addition, the ordering of the epochs is specified by their epoch number when the epochs are created, while with optimistic concurrency control the serial ordering is determined by the transaction commit order.

How does a programmer use this programming model to incrementally improve performance? First, and most importantly, the programmer can simply specify the decomposition of the transaction into epochs and do nothing further, and the result will be a correct parallel execution of the transaction (albeit one that does not necessarily perform very well). The



(a) Eliminating `*p` dependence can *hurt* performance.

(b) Execution with sub-epochs.

(c) Idealized parallel execution.

Figure 3: Sub-epochs improve performance when dependences exist.

epochs will likely have dependences between them which will cause violations (i.e., cause speculation to fail). If this is the case, then the programmer can use feedback from the TLS system to identify the cause of each dependence and try to eliminate them.

For example, one dependence which we encountered when parallelizing New Order was in the buffer pool: every time a page was requested and it was present in the buffer pool, the global variable `st_cache_hit` would be incremented. The increment would both read and write the variable, and so whenever an epoch requested a page from the buffer pool it would cause any later epochs which had already made a buffer pool request to be violated and restart. Once the system told us about this dependence it was easy to correct—we changed the code so that there was one copy of `st_cache_hit` per CPU, and updated the appropriate copy of the variable on a buffer pool hit. We then modified the code which reads this variable (which is only invoked rarely, since this variable is used only as a performance monitoring statistic) to sum all of the per-CPU copies. In this example a simple change to the code can eliminate a performance limiting data dependence.

2.1.1 Tolerating Dependence Violations

Previous TLS work has focused on programs with small epochs that have few frequent data dependences between them. Eliminating the only frequent data dependence between a pair of epochs removes the main source of violations, and results in a large performance gain. Unfortunately, in database systems the epochs are large and have many data dependences. How does eliminating a data dependence affect these epochs?

In Figure 3(a) we show what happens when the programmer eliminates the dependence caused by `p`. In this particular case, removing that dependence actually *hurts* performance! The problem is that when one dependence is eliminated it can expose other dependences, and merely delay an inevitable roll-back and re-execution. This problem can be avoided by dividing an epoch into several *sub-epochs* (Figure 3(b)). A sub-epoch acts like a checkpoint or sub-transaction: when a violation occurs execution is rolled back just to the last sub-epoch, which means that less work has to be rolled back and redone. Compare the execution with sub-epochs (Figure 3(b)) to an idealized parallel execution where each read does not execute until any dependent write has already executed (Figure 3(c)): as an epoch is broken into more and more sub-epochs, it approaches the performance of this idealized execution. In practice the number of supported sub-epochs is limited by hardware overheads [1], in this paper we assume 8 sub-epochs per epoch are used.

2.2 Hardware Support for TLS

The study presented in this paper is based on a hardware implementation of TLS which buffers speculative state in the cache hierarchy, and detects dependence violations by observing cache coherence messages [17, 5]. Previous TLS designs focused on applications with small epochs (50 to 5,000 dynamic instructions), while the large epochs (60,000 instructions) in a database transaction require the additional ability to buffer the state of very large epochs, as well as support for sub-epochs so that violations have a tolerable performance penalty. Therefore, concurrently with the work presented in this paper, we also designed a new implementation of TLS hardware that supports both large epochs and sub-epochs, the details of which are beyond the scope of this paper and are available in a technical report [1].

3 Techniques for Eliminating Dependences

To evaluate the potential of applying TLS to a database system, we implemented the New Order transaction from the TPC-C benchmark on the BerkeleyDB storage manager. We chose New Order as the representative transaction in TPC-C because it is rich in read and update operations, it accounts for half of the workload, and it is used to measure throughput in TPC-C (TMPC). Pseudo-code for the transaction is shown in Figure 1. We want to parallelize the `for` loop in the transaction, as this loop comprises 80% of execution time. At a high level, this loop reads the `item` table, updates the `stock` table, and appends to the `order_line` table. Since it is read-only, the read of the `item` table cannot cause a data dependence between epochs. The append to the `order_line` table should not cause data dependences either, since a new order line is generated for each epoch. The update of the `stock` table is potentially problematic—if two epochs were to refer to the same item, then one epoch's update of the `stock` table might interfere with another epoch's update. However, in the benchmark specification items are randomly chosen uniformly from a set of 100,000 items, and so the probability of any two epochs conflicting is very small. At this high level, thanks to the infrequency of data dependences, it appears that this loop is an ideal candidate for parallelization using TLS.

While our high-level analysis concludes that TLS parallelization is promising, the implementation details of query execution algorithms and access methods reveal more potentially-limiting data dependences: read/write accesses to locks, latches, the buffer pool, logging, and B-tree indexes will cause data dependences between epochs. To eliminate these data dependences we propose and analyze three techniques:

1. **Partition data structures.** A standard memory allocation operation (`db_malloc`) implementation uses a single pool of memory, and parallel accesses to this shared pool will conflict. Using a separate pool of memory for each concurrent epoch avoids such conflicts. Many other dependences are also due to multiple epochs sharing a resource in memory—these dependences can be avoided by partitioning that resource.

2. **Exploit isolated undoable operations (IUOs).** The TLS mechanism ensures that all attempts to fetch and pin a page(`pin_page`) in the buffer pool by one epoch complete before any invocations of `pin_page` in the next epoch begin, due to conflicts in the data structures which maintain LRU information. We prefer to allow `pin_page` operations to complete in any order. An epoch can simply call `pin_page` with speculation disabled: if the epoch is violated then the fetched page just remains in the buffer pool, and `unpin_page` can be invoked to release the page. This works because the `pin_page` operation is *undoable* and *isolated*. An undoable function has a corresponding function which can undo the isolated function's effects. An isolated function can be undone without affecting any other epoch or thread in the system. When speculatively parallelizing a transaction, we exploit isolated undoable operations by executing them non-speculatively, and call the corresponding undo function if the epoch is violated. This is similar to nested top actions in ARIES [12], since we modify the execution but preserve higher level semantics.

3. **Postpone operations until the end of the epoch.** When a log entry is generated, it is assigned a log sequence number and increments a global variable. This log sequence number counter forms a dependence between these two epochs. Our key insight was that an epoch never uses log sequence numbers—it only generates them. We can generate log entries during the execution of the epoch, and assign all of the sequence numbers at the end of the epoch after all previous epochs have completed, and just before committing the epoch (which makes the new log entries visible to the rest of the system). When an operation has no impact on the execution of the epoch, and instead only affects other transactions then it can be delayed until the end of the epoch.

In the next section we explore the major subsystems of the DBMS, and show how these three techniques can be used to eliminate the critical dependences we encountered while tuning the New Order transaction.

4 Performance Tuning

When we first parallelized the New Order transaction we encountered many dependences throughout the DBMS code. Some dependences are easy to eliminate through a local change to the source code: for example, false sharing dependences (see Section 4.5) can be eliminated by inserting padding in data structures so that independent variables do not share a single cache line. Other data dependences are inherent in the basic design of the database system, such as the creation of log sequence numbers or the locking subsystem. In the following sections we tour the database system's major components, and explain how we eliminate or avoid dependences on the common path in order to increase concurrency for TLS parallelization.

4.1 Resource Management

A large portion of every DBMS is concerned with resource management. Resources include latches, locks, cursors, private and shared memory, and pages in the buffer pool. All of these resources can be acquired and released. Dependences between epochs occur when two epochs try to acquire the same resource, or when the data structures which track unused resources are shared between epochs. In the following sections we look at each of these resources and develop strategies for executing them in parallel while avoiding data dependences.

4.1.1 Latches

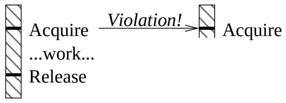
The database system uses latches extensively to protect data structures, and as a building block for locks. Latches are required for correct execution when multiple transactions are executing concurrently, and ensure that only one thread is accessing a given shared data structure at any time. Latches are typically held for only a short period of time—in Section 4.1.2 we discuss *locks*, which offer concurrency control for database entities.

Latches form a dependence between epochs because of how they are implemented: a typical implementation uses a read-test-write cycle on a memory location (which may be implemented as a test-and-set, load-linked/store-conditional, atomic increment, or other atomic instruction). This read-test-write cycle can cause a data dependence violation between epochs (Figure 4(a)).

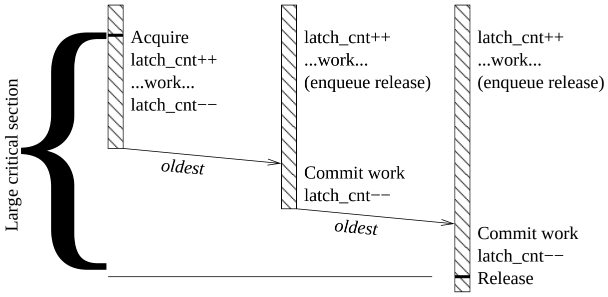
Under TLS, using latches to serialize accesses *within a transaction* is completely unnecessary—the TLS mechanism already ensures that any data protected by the latch is accessed in a serializable order, namely the original sequential program order. Hence using latches to preserve mutual exclusion between epochs is redundant with the existing TLS mechanism [8]. However, latches do ensure that mutual exclusion is maintained *between transactions*, and TLS does not perform that function. So we cannot simply discard the latches; we must instead ensure that they preserve mutual exclusion between transactions without causing violations between the epochs within a transaction.

There are two operations performed on a latch: *acquire* and *release*. Let us first consider release operations. When a latch is released, the latch and the data it protects become available to other transactions. Since, under TLS, modifications made by an epoch are buffered until it commits, we must postpone all release operations until after the epoch has fully committed. Release operations can be postponed by building a list of pending release operations as the epoch executes, and then performing all of the releases in the pending list when the epoch commits. If the epoch is violated, we simply reset this list.

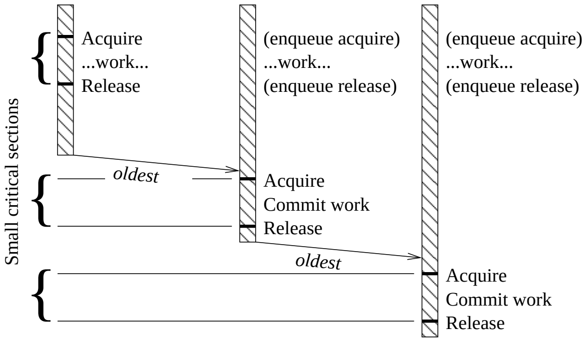
Next we consider acquire operations. During normal execution, when a latch is acquired it prevents other transactions in the system from changing the associated data. A naïve approach to handling a latch acquire under TLS is to perform the acquire *non-speculatively* at



(a) Latch operations create dependences.



(b) Aggressive latch acquire. The long critical section that results may cause performance issues.

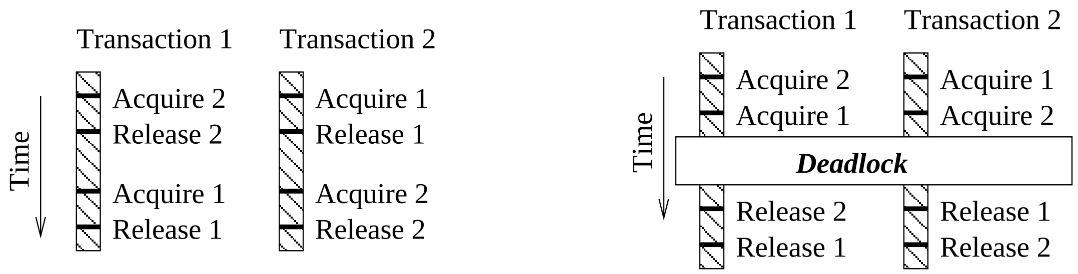


(c) Lazy latch acquire. Delaying the acquire shrinks the critical section.

Figure 4: Adapting latches for use under TLS execution.

the point when it is encountered. This can be implemented by a recursive latch, which counts the number of acquires and releases, and makes the latch available to other transactions only when the count reaches zero. This *aggressive* approach, shown in Figure 4(b), has a major drawback: since latch releases have been delayed until the end of the epoch, we have increased the overall size of the critical section. In addition, since we have parallel overlap between multiple critical sections in a single transaction, the latch may be held for an extended period of time.

To avoid long critical sections, we can also postpone acquires as shown in Figure 4(c). In this *lazy* approach, all latch acquires are performed at the end of the epoch, then the buffered speculative modifications are committed, and finally all latch releases are performed. This



(a) Latch operations before re-ordering.

(b) Latch operations after re-ordering.

Figure 5: Delaying latch release operations until after a epoch commits can introduce deadlock.

method results in much smaller critical sections, even when acquire and release operations for a given latch are encountered repeatedly during an epoch. A potential disadvantage of this approach is that if another transaction changes the protected data, the epoch will violate and restart.

Both the lazy and aggressive latch schemes have a potential problem: they re-order the latch release operations relative to the latch acquire operations as specified in the original program. If multiple latches are acquired by a single epoch, a deadlock may emerge that is not possible in the sequential execution, as shown in Figure 5. Although such deadlocks should be rare, there are two strategies to remedy them: *avoidance* and *recovery*. Deadlock can be avoided using two traditional techniques: (i) perform all latch acquires in a single atomic operation, or (ii) enforce a global latch acquire ordering [16], such as by sorting the acquire queue by latch address. If avoidance is not possible, we can instead recover from deadlock once detected (perhaps through a time-out) by violating and restarting one of the deadlocked epochs. Forward progress is guaranteed because there is always at least one epoch (the oldest) which executes non-speculatively. The key insight is that restarting an epoch is much cheaper than restarting the entire transaction since there are many epochs per transaction.

4.1.2 Locks

Locks are a more sophisticated form of concurrency control than latches. Instead of providing simple mutual exclusion, locks allow multiple threads into a critical section at the same time if the lock types are *compatible*: multiple readers are allowed into a critical section at a time, while writers have exclusive access. Locks also provide deadlock detection, since multiple locks can be held at once and they are meant to be held for longer periods of time than latches.

We start by parallelizing locks using a *lazy locking* scheme, similar to the *lazy latch* scheme in Section 4.1.1. When an acquire operation is encountered in speculative code, we cannot simply delay the entire acquire operation until the end of the epoch, since a handle must be returned. Instead, we return an *indirect* handle, which is a pointer to an empty handle that is filled in at the end of the epoch when the lock acquire is actually performed.

To summarize our scheme so far, at the end of an epoch all of the lock acquires encountered in that epoch are performed, the changes made by the epoch are committed, and then all of the lock releases encountered in the epoch are performed. This scheme will result in correct execution, but holding all of the locks used by an epoch *simultaneously* can be a performance bottleneck in the database, particularly for the locks used for searching B-trees. We avoid this problem through a minor change: at the end of the epoch we (i) acquire *and release* all *read-only* locks in the order that the acquire and release operations were encountered during the epoch, we then (ii) perform all *non-read-only* lock acquires that were encountered during the epoch, (iii) commit the epoch's changes to memory, and then (iv) perform all *non-read-only* lock releases that were encountered during the epoch. Since a B-tree search involves briefly acquiring a large number of read-only locks, this ensures that those locks are held for minimal time; we need not hold the *read-only* locks during the epoch commit because the system view of an epoch commit is similar to a transaction commit: it either succeeds or fails. By acquiring and releasing the locks we ensure that the epoch commit does not occur in the middle of a non-read-only critical section in some other transaction. If latches were labeled as read-only or read/write this optimization could also be applied to latches in addition to locks.

Our method of executing lock acquires may also possibly cause a deadlock situation. Similar to latches, we can recover from a detected deadlock situation by violating and restarting one of the deadlocked epochs.

4.1.3 Cursor Management

Cursors are data structures used to index into and traverse B-trees. Since they are used quite frequently and their creation is expensive, they are maintained in pre-allocated stacks. Unused cursors are stored in a *free cursor stack*. A dependence between epochs is created when one epoch puts a cursor onto the free cursor stack and the next epoch removes that cursor from the stack, since both operations manipulate the free pointer. Preserving this dependence is not required for correct execution: the second epoch did not need to get the exact same cursor, but instead wanted to get *any* cursor from the free stack. We can eliminate this dependence by partitioning the stack, and hence maintaining a separate stack for each processor. This implies that more cursors will have to be allocated, but that each cursor will only be used by the CPU which allocated it, increasing cache locality and eliminating dependences between epochs.

4.1.4 Memory Allocation

The free cursor pool mentioned above is just a special case of memory allocation. The general purpose memory allocators (such as `db_malloc`) in the database system introduce dependences between epochs when they update their internal data structures. To avoid these dependences, we must substitute an allocator designed with TLS in mind: in the common case, such an allocator should not communicate between CPUs. Fortunately, this is also a requirement of highly scalable parallel applications. The Hoard memory allocator [3] is one such allocator, which maintains separate free lists for each CPU, so that most requests for memory do not communicate. In the next section we show an even simpler way of avoiding dependences which does not require modifying or changing the underlying memory allocator.

4.1.5 Buffer Pool Management

When either a transaction or the DBMS itself need to read a page of the database, they request that page by invoking the `pin_page` operation on the buffer pool. This operation reads the requested page into memory (if it is not already there), pins it in memory, and returns a pointer to it. Once finished with the page, it is released by the `unpin_page` operation.

Conceptually, the buffer pool is very similar to the memory allocator, since it manages memory. However, the buffer pool is different because users explicitly name the memory they want, and different `pin_page` operations can pin the *same page*. Therefore, simply partitioning the page pool between epochs will not suffice. Instead, we exploit the fact that the order in which `pin_page` operations take place *does not matter.* If a speculative epoch fetches the wrong page from disk, we simply must return that page to the free pool. We implement this by executing the `pin_page` function non-speculatively, so that it really does get the page and pin it in a way which is visible to the entire system. If the epoch which called `pin_page` is later violated, we can undo this action by calling `unpin_page`. (This is similar to the compensating transactions used in Sagas [4].)

The code wrapper shown in Figure 6 implements this modified version of `pin_page`. In particular, this code does the following.

1. Provides thorough error checking. Since this routine is called from a speculative thread, the parameters could be invalid.

2. Acquires a latch which provides mutual exclusion between epochs within a transaction, to guard against the possibility that `pin_page` was not implemented with intra-transaction concurrency in mind. However, this latch can be eliminated if the programmer determines that the implementation of `pin_page` is safe in this respect.

3. Temporarily suspends speculation. While speculation is suspended, the epoch is *non-speculative* and hence all reads will observe committed machine state and all writes will be immediately visible to the rest of the system (i.e., no buffering occurs). Hence reads performed by `pin_page` will not cause violations.

4. Saves a pointer to the recovery function, `unpin_page`. If the epoch is violated then `unpin_page` will be called to undo the execution of the corresponding call to `pin_page`.

Relaxing ordering constraints simplifies coding: instead of redesigning the buffer pool to be amenable to TLS execution, we place this simple wrapper around the allocation function.

```
page_t *pin_page_wrapper(pageid_t id) {
    static intra_transaction_latch mut;
    page_t *ret;

    suspend_speculation();                  // (3)
    check_pin_page_arguments(id);           // (1)
    acquire_latch(&mut);                    // (2)

    ret = pin_page(id);

    release_latch(&mut);                    // (2)
    on_violation_call(unpin_page, ret);     // (4)
    resume_speculation();                   // (3)

    return ret;
}
```

Figure 6: Wrapper for the `pin_page` function which allows the ordering between epochs to be relaxed.

However, this method requires that the `pin_page` function be an *isolated undoable operation.* The `pin_page` function is undoable: calling `unpin_page` undoes the call to `pin_page`. The `pin_page` is also isolated: when it is undone via `unpin_page` no other transaction or earlier epoch is forced to rewind or otherwise alter its execution.

Similar reasoning shows that the cursor allocation function and `db_malloc` are also isolated undoable operations, and so this code template could be applied to these functions instead of partitioning their free pools. The `pin_page` and `latch_acquire` functions also look like isolated undoable operations—but as we found above in Section 4.1.1, without great care speculatively executing these functions out of original sequential order can cause performance problems (by increasing critical section sizes) or create deadlock conditions (by re-ordering latch acquires).

The `unpin_page` operation for the buffer pool is *not* undoable, since an attempt to undo it with a `pin_page` operation may cause the page to be mapped at a different address. Because of this, we treat it similarly to a lock or latch release operation, and enqueue it to be executed after the epoch commits.

4.2 The Log

Every time the database is modified the changes are appended to the log. For recovery to work properly (using ARIES [12]) each log entry must have a log sequence number. Unfortunately, incrementing the log sequence number causes a data dependence between epochs. To avoid this dependence, we modify the logging code to append log entries for speculative epochs to a per-CPU buffer. When an epoch commits, we loop over this buffer to assign log sequence numbers to log entries, then append the entire buffer to the log.

4.3 B-Trees

B-trees are used extensively in the database system to index the database. The primary operations involving the B-tree are reading records, updating existing records, and inserting new records. Neither reading nor updating records modify the B-tree, and hence will not cause dependences between epochs. In contrast, insert operations modify the leaf pages of the B-tree. Therefore if the changes made by two epochs happen to fall on the same page then the update of the free space count for that page can cause a violation.

One strength of TLS parallelization is that infrequent data dependences need not be addressed, since the TLS mechanism will ensure correctness in such cases. An example of such an infrequent data dependence is a B-tree page split. Page splits can also cause lots of data dependences, but since they happen infrequently (by design) we can afford to just ignore them. In the rare cases when they occur, the TLS mechanism will ensure their correct sequential execution. The TLS mechanism provides a valuable fall back, allowing the programmer to avoid the effort of designing a algorithm for parallel page-splits.

The B-tree code in BerkeleyDB contains a simple performance optimization: when a search is requested, it begins the search by inspecting the page located by the previous search through a "last page referenced" pointer (this assumes some degree of locality in accesses). Accesses to this pointer cause a data dependence between epochs. Since the resulting violations can hurt performance, we decided to disable this "last page" optimization for TLS execution. Alternatively, one could retain this optimization without causing violations by maintaining a separate "last page reference" pointer per CPU.

4.4 Error Checks

Our study indicated that error checking code in the database system can occasionally cause dependences between epochs. The most important of these is a dependence caused by reference counting for cursors—a mechanism in the DBMS which tracks how many cursors are currently in use by a transaction, and ensures that none are in use when the transaction commits. Since this code is solely for debugging a transaction implementation, it can be safely removed once the transaction has been thoroughly tested.

4.5 False Sharing

To minimize overhead, the TLS mechanism tracks data dependences at the granularity of a cache-line. However, accesses to different variables which happen to be allocated on the same cache line can cause data dependence violations due to *false sharing*. This problem can be remedied by inserting padding to ensure that variables which are frequently-accessed by different CPUs are not allocated on the same cache line.[1]

[1]Insertion of padding works for most data structures, but is not appropriate for data structures which mirror disk-resident data, such as B-tree page headers. In this case, changes will have to be made to the

5 Experimental Results

In this section we evaluate both the ease with which a DBMS programmer can parallelize a representative transaction, as well as the resulting performance of that transaction.

5.1 Benchmark Infrastructure

Our experimental workload executes the New Order transaction running on top of BerkeleyDB 4.1.25. Evaluations of techniques to increase concurrency in databases typically configure TPC-C to use multiple warehouses, since transactions would quickly become lock-bound with only one warehouse. In contrast, our technique is able to extract concurrency from within a single transaction, and so we configure TPC-C with only a single warehouse. The parameters for each New Order transaction are chosen using the Unix `random` function, and each experiment uses the same seed for repeatability. The benchmark executes as follows: (i) start the DBMS; (ii) execute 10 New Order transactions to warm up the buffer pool; (iii) start timing; (iv) execute 100 New Order transactions; (v) stop timing.

All code is compiled using `gcc` 2.95.3 with O3 optimization on a SGI MIPS-based machine. The BerkeleyDB database system is compiled as a shared library, which is linked with the benchmark that contains the transaction code.

To apply TLS to this benchmark we start with the unaltered (New Order) transaction, mark the main loop within it as parallel, and executed it on a simulated system with TLS support. The system reports back the load and store program counters of the instructions which caused violations, and we use that information to determine the cause (in the source code) of the most critical performance bottleneck. We then apply the appropriate optimization from Section 4 and repeat.

5.2 Simulation Infrastructure

We perform our evaluation using a detailed, trace-driven simulation of a chip-multiprocessor composed of 4-way issue, out-of-order, superscalar processors similar to the MIPS R14000 [19], but modernized to have a 128-entry reorder buffer. Each processor has its own physically private data and instruction caches, connected to a unified second level cache by a crossbar switch. Register renaming, the reorder buffer, branch prediction, instruction fetching, branching penalties, and the memory hierarchy (including bandwidth and contention) are all modeled, and are parameterized as shown in Table 1. Latencies due to disk accesses are not modeled, and hence these results are most readily applicable to situations where the database's working set fits into main memory.

5.3 Scaling Intra-Transaction Parallelism

In the Normal bars of Figure 7 we see the performance of the optimized transaction as the number of CPUs is varied. The Seq bar represents the unmodified benchmark running

B-tree data structure itself (see Section 4.3).

Table 1: Simulation parameters.

Pipeline Parameters	
Issue Width	4
Functional Units	2 Int, 2 FP, 1 Mem, 1 Branch
Reorder Buffer Size	128
Integer Multiply	12 cycles
Integer Divide	76 cycles
All Other Integer	1 cycle
FP Divide	15 cycles
FP Square Root	20 cycles
All Other FP	2 cycles
Branch Prediction	GShare (16KB, 8 history bits)

Memory Parameters	
Cache Line Size	32B
Instruction Cache	32KB, 4-way set-assoc
Data Cache	32KB, 4-way set-assoc,2 banks
Unified Secondary Cache	2MB, 4-way set-assoc, 4 banks
Speculative Victim Cache	64 entry
Miss Handlers	128 for data, 2 for insts
Crossbar Interconnect	8B per cycle per bank
Minimum Miss Latency to Secondary Cache	40 cycles
Minimum Miss Latency to Local Memory	75 cycles
Main Memory Bandwidth	1 access per 20 cycles

on a single core of an 8 core chip multiprocessor, while the 2 CPU, 4 CPU and 8 CPU bars represent the execution of full TLS-optimized executables running on 2, 4 and 8 CPUs. Large improvements in transaction latency can be obtained by using 2 or 4 CPUs, although the additional benefits of using 8 CPUs are small.

To better understand this data we break down each bar by where time is being spent—the breakdown is explained in Table 2. In Figure 7 we have normalized all bars to the 8 CPU case so that the subdivisions of each bar can be directly compared. This means that the Seq breakdown shows one CPU executing and 7 CPUs idling, the 2 CPU breakdown shows two CPUs executing and 6 CPUs idling, and so on.

None of the bars show that a significant fraction of the time was spent on failed speculation—this means that our performance tuning was successful at eliminating performance critical data dependences. As the number of CPUs increases there is a nominal increase in both failed speculation and time spent awaiting the latch used to serialize isolated undoable operations: as more epochs are executed concurrently, contention increases for both shared data and the latch. As the number of CPUs increases there is also an increase in time spent awaiting cache misses: spreading the execution of the transaction over more CPUs decreases

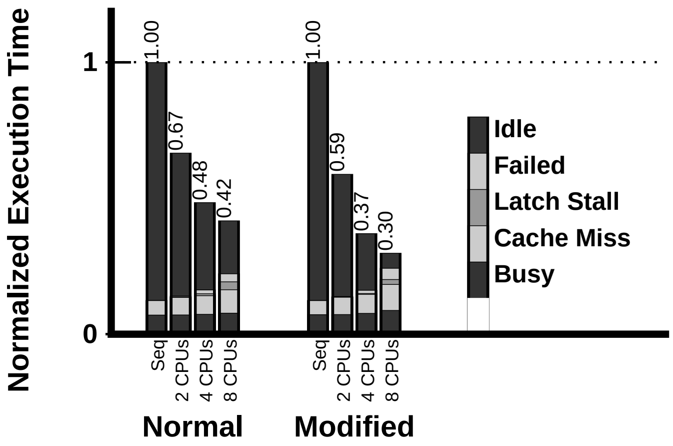


Figure 7: Performance of optimized benchmark while varying the number of CPUs.

Table 2: Explanation of graph breakdown.

Category	Explanation
Idle	Not enough threads were available to keep the CPUs busy.
Failed	CPU executed code which was later undone due to a violation (includes all time spent executing failed code.)
Latch Stall	Stalled awaiting latch; latch is used in isolated undoable operations.
Cache Miss	Stalled on a cache miss.
Busy	CPU was busy executing code.

cache locality, since the execution is partitioned over more level 1 caches.

The dominant component of all of the NORMAL bars in Figure 7 is *idle* time, for three reasons. First, in the SEQ, 2 CPU and 4 CPU case we show the unused CPUs as idle. Second, the loop that we parallelized in the transaction only covers 80% of the transaction's execution time, and during the remaining 20% of the time only one CPU is in use. Third, TPC-C specifies that the NEW ORDER transaction will order between 5 and 15 items, which means that on average each transaction will have only 10 epochs—this means that as we execute the last epochs in the loop load imbalance will leave CPUs idling. The effects of all three of these issues are magnified as more CPUs are added. To reduce idle time we modified the invocation of the transaction so that each order contains between 50 and 150 items (shown in the MODIFIED bars of Figure 7). We found that this modification decreases the amount of time spent idling, and does not significantly affect the trends in cache usage, violations, or idle time. The MODIFIED bars demonstrate that transactions which contain more parallelism make more effective use of CPUs.

The results in Figure 7 show that there is a performance trade-off when using TLS to

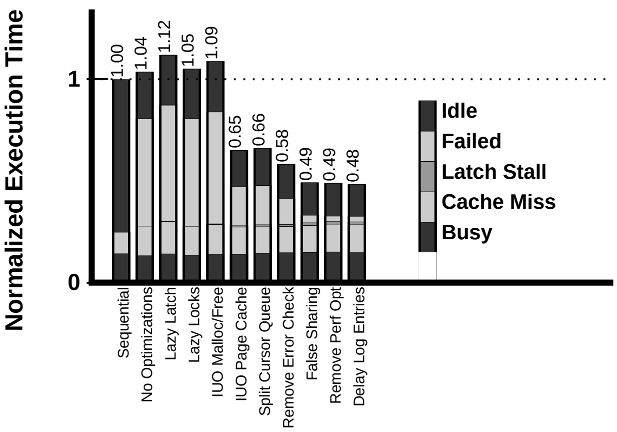


Figure 8: Performance impact of adding each optimization one-by-one on a four CPU machine.

exploit intra-transaction parallelism: devoting more CPUs to executing a single transaction improves performance, but there are diminishing returns due to a lack of parallelism, increased contention, and/or a decrease in cache locality. One of the strengths of using TLS for intra-transaction parallelism is that it can be enabled or disabled at any time, and the number of CPUs can be dynamically tuned. The database system's scheduler can dynamically increase the number of CPUs available to a transaction if CPUs are idling, or to speed up a transaction which holds heavily contended locks. If many epochs are being violated, and thus the intra-transaction parallelism is providing little performance benefit, then the scheduler could reduce the number of CPUs available to the transaction. If the transaction compiler simply emitted a TLS parallel version of *all* loops in transactions then the scheduler could use sampling to choose loops to parallelize: the scheduler could periodically enable TLS for loops which are not already running in parallel, and periodically disable TLS for loops which are running in parallel. If the change improves performance then it is made permanent.

5.4 Impact of Each Optimization

In Figure 8 we see the results of the optimization process on a four CPU system. In this case the breakdown of the bars is normalized to a four CPU system, and so $\frac{3}{4}$ of the SEQUENTIAL bar is marked as *Idle*, since three of the four CPUs are idling during the entire execution. The NO OPTIMIZATIONS bar shows what happens if we parallelize the loop and make no other optimizations—the existing data dependences in the DBMS prevent any parallelism from being exploited, and the fact that we have taken a sequential transaction and run it on four CPUs has reduced cache locality, causing it to slow down slightly.

The major source of violations in our newly-parallelized transaction are the reads and writes to latches; hence we perform the lazy latch optimization described in Section 4.1.1. This optimization fixes the first performance bottleneck, and exposes the next bottleneck which is in the lock code. The first optimization also causes a bit of a slowdown, since the next bottleneck causes violations to happen further away from sub-epoch boundaries—hence when violations do occur more execution has to be rewound (illustrated in Figure 3).

Once we have eliminated latches as a bottleneck, the next bottleneck exposed is in the locking subsystem. We remove the lock bottleneck by implementing lazy locks from Section 4.1.2. We continue to remove the bottlenecks one by one: applying the code template from Figure 6 to `db_malloc` and the `pin_page` operation, parallelizing the free cursor pool, removing dependence causing error checks (Section 4.4), adding padding to avoid violations due to false sharing (Section 4.5), removing the "last page referenced" pointer from the B-tree search code (Section 4.3), and delaying the generation of log entries until epochs are ready to commit (Section 4.2).

It is tempting to look at Figure 8 and conclude that the most important optimization was parallelizing the buffer pool, since adding this optimization caused the execution time to drop by 40%. However, this is not the case since the impact of the optimizations is *cumulative*. If we take the No Optimizations build and just enable the buffer pool optimization then the normalized performance is 0.98. Instead, Figure 8 implies that the iterative optimization process which we used works well—as the DBMS programmer removes performance limiting dependences performance gradually improves (and exposes new dependences). Removing dependences decreases the time spent on failed execution, and improves performance.

6 Conclusions

Chip multiprocessing has arrived, as evidenced by recent products (and announced road maps) from Intel, AMD, IBM and Sun Microsystems. While the database community has long embraced parallel processing, the fact that an application *must* exploit parallel threads to tap the performance potential of these additional CPU cores presents a major challenge for desktop applications. Processor architects have responded to this challenge through a new mechanism—*thread-level speculation* (TLS)—that enables optimistic parallelization on chip multiprocessors. Fortunately for the database community, although TLS was originally designed to overcome the daunting challenge of parallelizing desktop applications, it also allows us to tap new forms of parallelism within a DBMS that had previously been too painful to consider.

In this paper, we have focused on one such opportunity enabled by TLS: exploiting *intra-transaction* parallelism. Our experimental results demonstrate that we can speed up the *latency* (not just the throughput) of the most important transaction in TPC-C (New Order) by more than twofold by exploiting TLS on a chip multiprocessor with four CPU cores, or by a factor of 1.5 with two cores. TLS allows the database's scheduler to use CPU cores to improve latency when throughput is not the primary concern. In contrast with previous approaches to exploiting intra-transaction parallelism, we place almost no burden on the transaction programmer (they merely demarcate epoch boundaries). Although changes to the DBMS code are required to achieve this benefit, they affected less than 1200 out of 180,000 lines of code in BerkeleyDB, they were implemented in roughly a month by a graduate student, and we expect that they would generalize to other DBMSs. We hope that these promising results will inspire database researchers to find other opportunities for exploiting untapped parallelism through TLS.

References

[1] C.B. Colohan, A. Ailamaki, J.G. Steffan, and T.C. Mowry. Extending Thread Level Speculation Hardware Support to Large Epochs: Databases and Beyond. Technical Report CMU-CS-05-109, School of Computer Science, Carnegie Mellon University, March 2005.

[2] IBM Corporation. *IBM DB2 Universal Database Administration Guide: Performance.* IBM Corporation, 2004.

[3] E.D. Berger and K.S. McKinley and R.D. Blumofe and P.R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the 9th ASPLOS*, 2000.

[4] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259. ACM Press, 1987.

[5] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative Versioning Cache. In *Proceedings of the 4th HPCA*, February 1998.

[6] J. Gray. *The Benchmark Handbook for Transaction Processing Systems.* Morgan-Kaufmann Publishers, Inc., 1993.

[7] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro Magazine*, March-April 2000.

[8] M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th ISCA*, 1993.

[9] H. Kaufmann and H. J. Schek. Extending tp-monitors for intra-transaction parallelism. In *Proceedings of the 4th PDIS*, 1996.

[10] H.T. Kung and J.T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, pages 213–226, June 1981.

[11] J.H. Miller and H. Lau. *Microsoft SQL Server 2000 Resource Kit*, chapter RDBMS Performance Tuning Guide for Data Warehousing, pages 575–653. Microsoft Press, 2001.

[12] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM TODS*, March 1992.

[13] Michael Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In *Proceedings of the Summer Usenix Technical Conference*, June 1999.

[14] M.K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *Proceedings of PPoPP*, June 2003.

[15] M. Rys, M.C. Norrie, and H.J. Schek. Intra-transaction parallelism in the mapping of an object model to a relational multi-processor system. In *Proceedings of the 22nd VLDB*, 1996.

[16] A. Silberschatz, P.B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., 2002.

[17] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *ISCA 27*, June 2000.

[18] M. Tremblay. MAJC: Microprocessor Architecture for Java Computing. *HotChips '99*, August 1999.

[19] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, April 1996.

[20] C. Zuzarte. Personal communication, 2005.