Incrementally Parallelizing Database Transactions with Thread-Level Speculation

CHRISTOPHER B. COLOHAN
Carnegie Mellon University, Google, Inc.
ANASTASSIA AILAMAKI
Carnegie Mellon University
J. GREGORY STEFFAN
University of Toronto
and
TODD C. MOWRY
Carnegie Mellon University, Intel Research Pittsburgh

With the advent of chip multiprocessors, exploiting intratransaction parallelism in database systems is an attractive way of improving transaction performance. However, exploiting intratransaction parallelism 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 article 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 incrementally parallelizing transactions, we can dramatically improve performance: on a simulated four-processor chip-multiprocessor, we improve the response time by 44–66% for three of the five TPC-C transactions, assuming the availability of idle processors.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Concurrency, transaction processing*; C.1.4 [**Processor Architectures**]: Parallel Architectures; D.3.4 [**Programming Languages**]: Processors—*Compilers*

This research has been supported by grants from the National Science Foundation, NASA, IBM, and Intel.

Authors' addresses: C. B. Colohan, Google, Inc., Mountain View Campus, 1600 Ampitheatre Parkway, Mountain View, CA 94043; email: colohan@google.com; A. Ailamaki and T. Mowry, Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213; email: natassa@cmu.edu, tcm@cs.cmu.edu; J. G. Steffan, Dept. of Electrical and Computer Engineering, Computer Engineering Research Group, University of Toronto, 10 King's College Road, Toronto, ON, Canada M5S 3G4; email: steffan@eecg.toronto.edu.

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Thread-level speculation, optimistic concurrency, chip-multiprocessing, incremental parallelization

ACM Reference Format:

Colohan, C. B., Ailamaki, A., Steffan, J. G., and Mowry, T. C. 2008. Incrementally parallelizing database transactions with thread-level speculation. ACM Trans. Comput. Syst. 26, 1, Article 2 (February 2008), 50 pages. DOI = 10.1145/1328671.1328673 http://doi.acm.org/10.1145/1328671.1328673

1. INTRODUCTION

We are in the midst of a major change in microprocessor design: current and future processors will be multicore, with the number of cores per chip potentially doubling every silicon generation. How can database systems exploit this increasing abundance of hardware-supported threads? Currently, for *online transaction processing* (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 *intraquery 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 *intratransaction parallelism* [IBM Corporation 2004; Miller and Lau 2001; Zuzarte 2005], and for good reason.

Parallelizing a transaction is difficult. First, the DBMS must be modified to support multiple threads per transaction. Latches (a.k.a. mutexes) must be added to data structures that 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, the prevalent transaction in TPC-C [Gray 1993] (Figure 1), that simulates the workload of a database server owned by a bank or business selling widgets. We can parallelize the main loop (which represents 78% of the execution time) such that each loop iteration runs as a thread. The transaction programmer must understand when these threads may share data, and add interthread locks to avoid problems; for example, the thread should use interthread 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.

The difficulty in parallelizing a transaction lies in *data dependences* between threads. When any two threads may share data, synchronization must be added to preserve program semantics. By adding locks and latches into the source code, the programmer is letting the system know what invariants must hold to compute the correct result. But the programmer must be conservative, and protect against all *possible* sharing patterns. The challenge with this conservative programming model is that the programmer must add latches and locks to

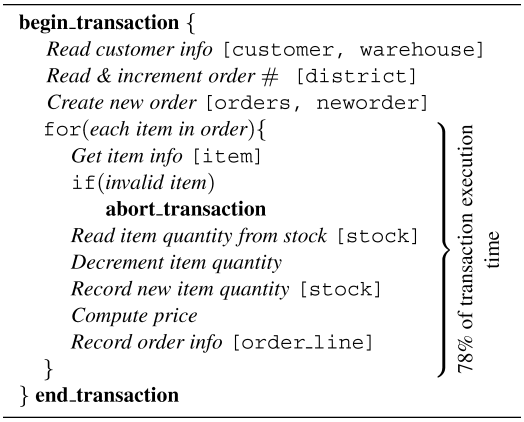


Fig. 1. The NEW ORDER transaction. In brackets are the database tables touched by each operation.

preserve correctness: if the programmer does not have a deep understanding of the entire program, then they will likely either unknowingly introduce bugs by omitting essential locks, or else degrade performance by adding unnecessary locks. Instead we would prefer a programming model that allows the programmer to focus on improving performance rather than worrying about affecting correctness.

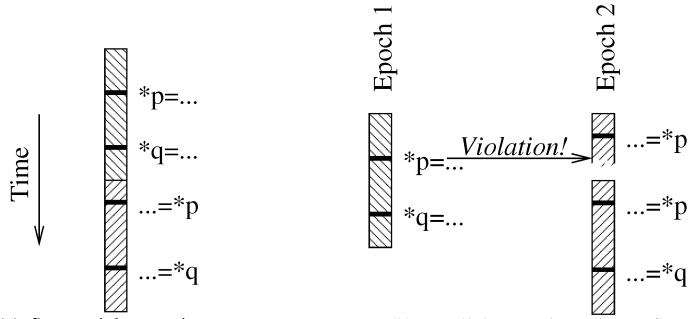
1.1 Incremental Parallelization with Thread-Level Speculation

Thread-level speculation (TLS) [Gopal et al. 1998; Hammond et al. 2000; Steffan et al. 2000; Tremblay 1999; Zhang et al. 1999] empowers the programmer to speculatively parallelize code while being concerned only with performance rather than correctness. With TLS, the programmer specifies where to break a transaction into threads, or *epochs*,[1] and the TLS mechanism executes them in parallel while preserving the original sequential semantics of the program. The TLS mechanism preserves sequential semantics by tracking data dependences between epochs and restarting epochs when their execution diverges from the original sequential execution.

Figure 2 illustrates the basic operation of TLS, where sequential code (Figure 2(a)) is divided into *epochs* that 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 that has already been read by epoch 2—if so, then epoch 2 has *violated* sequential semantics, and is rewound and reexecuted with the correct value.

This article describes our experience in incrementally parallelizing database transactions using TLS—in particular, how small changes can be made to the DBMS to avoid the most problematic data dependences between epochs. The

[1]We refer to the parallel threads in TLS as *epochs* to differentiate them from the explicit threads that may also exist in the system. For example, in a database system each transaction runs on a thread, and we break each of those transaction threads up into epochs.



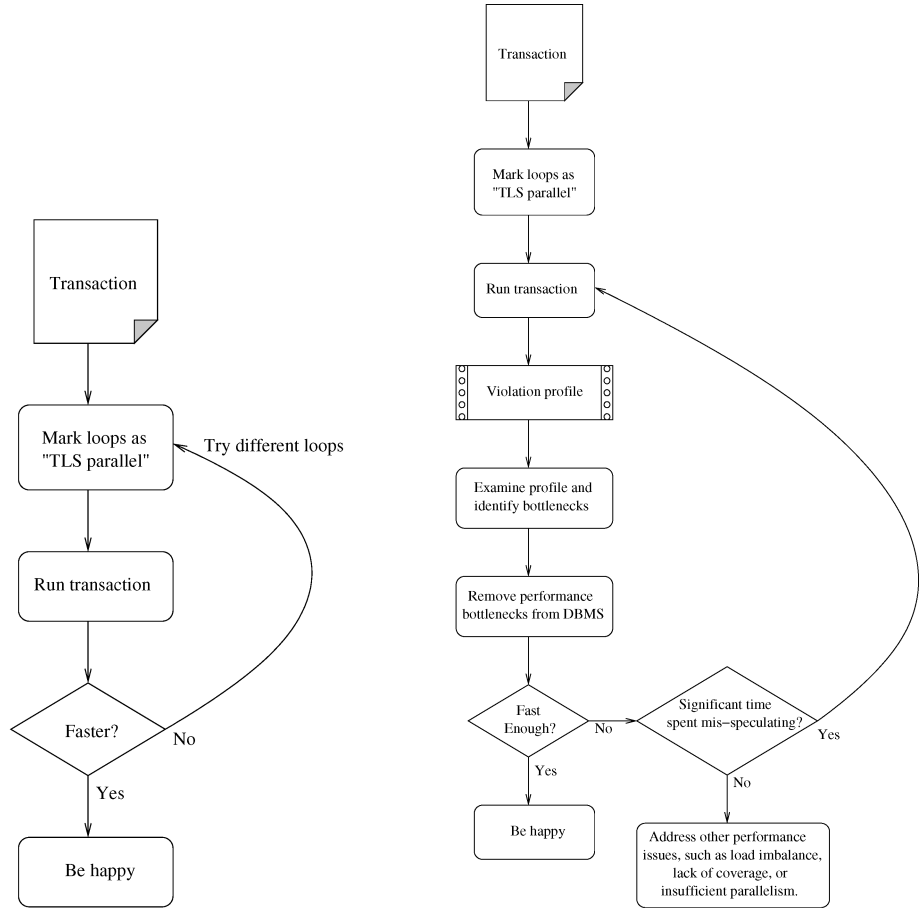
(a) Sequential execution. (b) Parallel execution with TLS.

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

result is a significant latency improvement for database transactions—on a simulated four-processor chip-multiprocessor, the response time is improved by 46–66% for three of the five TPC-C transactions. To achieve these performance benefits, we need to modify both the transaction code and the code in the DBMS.

Transaction programmers are primarily concerned with creating functional applications. We do not want to teach transaction programmers new sophisticated programming techniques, nor do we want to introduce new failure modes into their applications. We let the transaction programmer treat TLS as a switch. This switch can be enabled or disabled for any loop in the transaction, and the only effect visible to the transaction programmer is that TLS may provide improved performance when TLS is enabled (Figure 3(a)). TLS can extract parallelism from within a transaction without changing the original sequential execution semantics, meaning that TLS will not introduce any new bugs (although the change in performance may uncover hidden race conditions that already existed in the original code). The interface to TLS used by the transaction programmer can be as simple as providing performance hints without having to modify the transaction software.

Database vendors want to provide high performance database systems to customers. At the same time, existing database systems are very large and complex systems, and so the database system programmers are reluctant to make changes that modify large portions of the database system. To apply TLS, the database system programmer engages in an iterative process (Figure 3(b)): starting with a sequential transaction, the DBMS programmer marks some loops for TLS execution, and then examines the execution profile provided by the hardware. This profile information indicates which loads and stores in the transaction caused dependence violations that in turn reduced performance. The DBMS programmer then optimizes *just this performance critical code*, and repeats the process. In this work we modified the BerkeleyDB DBMS [Olson et al. 1999] to learn how much work the DBMS programmer (vendor) would have to go through to support TLS. Although BerkeleyDB is small compared to commercial DBMS systems, it contains many of the complexities found in all DBMS systems. This tuning process required modifying only a small fraction of the BerkeleyDB DBMS code (we only modified 1200 out of 180,000 lines of code), and yet eliminated the majority of the data dependences that cause violations.



(a) Transaction programmer work flow. (b) Database system programmer work flow.

Fig. 3. How the transaction programmer and database system programmer use TLS.

Hence the gains of TLS may be obtainable through a small effort on the part of the database vendor.

1.2 Why Transaction Latency?

You may believe that transaction throughput is all that matters for database systems, and in particular for OLTP workloads. Why are we attacking transaction latency? First, there is the obvious reason to improve latency: improving latency makes users happy by improving system response time. The second reason is more subtle: on modern systems, OLTP workloads such as TPC-C are frequently lock-bound [McWherter et al. 2004]. If the system is lock-bound, then to improve performance we need to decrease the time spent waiting for locks. Although we do not show this experimentally in this article, we believe that TLS can be used to improve the latency of a transaction that holds contended locks, and hence the transaction will release those locks more quickly—resulting in an improvement in overall transaction throughput [McWherter et al. 2005].

1.3 Related Work

The previous and concurrent work which inspired and influenced our research can be divided into three areas: research in intratransaction parallelism, research in TLS software support, and research in TLS hardware support. We now discuss each of these in greater detail.

1.3.1 *Intratransaction Parallelism.* Traditionally, high-performance database systems have targeted intertransaction parallelism, or intraoperation parallelism, while this article introduces new techniques for exploiting intratransaction parallelism. Previous work on intratransaction parallelism has focused on techniques which do not require modifying the DBMS: with *Sagas* the programmer is able to define a long-running transaction, known as a *saga*, which is composed of several DBMS-visible transactions. By using sagas a long transaction could execute without holding locks for an extended period of time. To allow a saga to abort the transaction, the programmer would have to create *compensating transactions* which undo the side-effects of the individual transactions within the saga. TLS is complimentary to sagas—TLS can be used *instead* of sagas to allow a long running transaction to complete faster (and release its locks faster); TLS can also be used *in addition to* sagas to improve the response time of the individual transactions within a saga. TLS is easier to use since the hardware automates the restarting of epochs, meaning that when using TLS there is no need for the transaction programmer to write *compensating epochs* to allow epochs to abort.

The work on sagas evolved into work on TP-Monitors [Kaufmann and Schek 1996; Rys et al. 1996], which coordinate the execution of the transactions within a saga and allow some of those transactions to execute in parallel, improving performance further. This resulted in work by Shasha et al. [1995], who developed a theoretical basis for automatically breaking a saga into transactions. Shasha et al. [1995] showed that, if a conflict graph can be constructed for a set of transactions, then the transactions can be *chopped* into smaller transactions which increases the degree of concurrency in the workload. The conflict graph is a static analysis, which allows the transactions within a saga to execute in parallel in the absence of any possible dependences between them. In contrast, TLS is more optimistic: TLS allows epochs to execute in parallel, and only restarts epochs when dependences actually occur. As a result, TLS is able to exploit more parallelism, since it takes advantage of dependence information which is only available at runtime.

1.3.2 *TLS Software Support.* Most previous TLS research has endeavored to have the *compiler* automatically parallelize programs [Ooi et al. 2001; Bhowmik and Franklin 2002; Vijaykumar 1998; Johnson et al. 2004]. This compiler work grew out of the pioneering work of Knight and Halstead, who developed functional languages with support for TLS-style execution [Knight 1986; Halstead 1985]. In this work our goal is to parallelize database transactions. Database transactions and the DBMSs they utilize are not written in functional languages, and it is not practical to reimplement them in functional languages. Compiler-based techniques tend to assume perfect knowledge about the whole

program to be parallelized, while with databases the transaction is usually compiled separately from the DBMS, and the interface between the transaction and DBMS is kept deliberately simple for portability reasons. We found that transactions naturally decompose into threads which are much larger and more complex than the threads generated by compiler based techniques.

In this article we use hardware-assisted speculative execution to simplify manual parallelization. Prabhu and Olukotun showed that using TLS to assist in manual parallelization has great promise, since it works well when applied to SPEC benchmarks on the Hydra multiprocessor [Prabhu and Olukotun 2003; Hammond et al. 2000]. These results encouraged us to apply TLS to the much larger epochs from the outer loops of database transactions.

Hammond et al. pushed the idea of programming with epochs to its logical extreme, making the program consist of nothing but epochs; this results in a simpler architecture [Hammond et al. 2004b], but requires the programmer to always use epochs [Hammond et al. 2004a]. We believe that it is not practical to apply this approach to existing database systems. As a result the work in this article does not enforce a single programming model: programmers can use TLS and epochs when they are desirable, and use either traditional sequential execution or threaded execution as well.

1.3.3 *TLS Hardware Support.* The basic idea behind TLS is inspired by Kung and Robinson's [1981] optimistic concurrency control (OCC) work, an alternative to using locks in databases. Kung and Robinson proposed a mode of execution similar to TLS's epochs—transactions execute without using locks, and before committing a check for conflicts is performed, and the transaction rewinds if a conflict is detected. OCC was implemented in software, and achieved reasonable performance by considering only dependences caused by accesses to the database itself. TLS tracks dependences caused by accesses to both the database and the meta-data used to maintain the database. This has two benefits: first TLS is able to extract parallelism from within a single transaction (instead of increasing parallelism between transactions). This means that when speculation fails less work is undone, since only a fragment of the transaction's execution is rewound. Second, this article shows that the changes to the DBMS software required for achieving good performance with TLS are localized, while applying OCC requires changing the fundamental locking methodology used by the entire DBMS.

The optimistic concurrency control work inspired a hardware implementation called *transactional memory* [Herlihy and Moss 1993], which showed how the processor caches can be used to buffer speculative state. The transactional memory work led to a tech report which did a preliminary investigation of TLS-style execution [Morrisett and Herlihy 1993]. The hardware design in this article builds on this idea, using the caches to buffer speculative state.

The first major study of how to implement a complete TLS system in hardware was the Multiscalar project from Wisconsin [Franklin and Sohi 1996; Sohi et al. 1995]. The initial Multiscalar featured an architecture optimized purely for TLS-style execution. Programs were broken up into *tasks* (equivalent to TLS epochs), and each task was run on a separate CPU. Register dependences were

handled through a fast register forwarding ring, and memory dependences were resolved through a centralized address resolution buffer [Franklin and Sohi 1996]. Later this design was refined to use the caches to detect and buffer memory dependences, in the form of the speculative versioning cache [Gopal et al. 1998]. The success of the Multiscalar project inspired numerous other TLS research projects, including the IACOMA project [Prvulovic et al. 2001], the Hydra project [Hammond et al. 2000], and our Stampede project [Steffan and Mowry 1998; Steffan et al. 1997, 2000, 2002; Zhai et al. 2002, 2004]. This research also inspired work on a few software-only TLS designs [Gupta and Nim 1998; Rauchwerger and Padua 1999; Rundberg and Stenstrom 2000] and hardware-only TLS designs [Akkary and Driscoll 1998; Marcuello and González 1999; Rotenberg et al. 1997]. An interesting comparison of many of these schemes was done by Garzarán et al. [2003].

An alternative to increasing intratransaction parallelism is to attempt to increase intertransaction parallelism, which could also be done with the help of TLS hardware to implement OCC. In particular, Rajwar and Goodman [2001] proposed a hardware implementation allowing the speculative elision of locks in parallel programs, and Martínez and Torrellas [2002] proposed using TLS to support speculative synchronization.

One of the ways in which we avoid dependences in this article is to delay lock and latch operations during the execution of an epoch, and optimistically assume there will not be conflicts before the epoch commits. There are two differences between this technique and optimistic concurrency control [Kung and Robinson 1981; Herlihy and Moss 1993]: (i) epochs are much smaller than transactions (in our experiments we have between 2 and 192 epochs per transaction), and (ii) transactions using speculation in our TLS scheme are able to correctly interact with nonspeculative transactions with *no* changes to the nonspeculative transactions.

This article adds an important capability to prior hardware designs: we use subepochs [Colohan et al. 2006] to tolerate data dependences between speculative epochs. Subepochs (a.k.a. *subthreads*) are a form of checkpointing, and in this article subepochs are used to reduce the penalty due to failed speculation. Using checkpoints in epochs was previously proposed by Olukotun et al. [1999]—in their work they found that checkpoints had little benefit, since they were considering workloads with small epochs. The large epochs found in our database workloads improve dramatically with the use of subepochs (checkpoints).

1.4 Contributions

This article makes the following contributions: (i) it provides a possible solution to the problem of parallelizing the central loop of a transaction that can reduce transaction latency and hence decrease contention for resources used by the transaction; (ii) it provides a methodology for eliminating the data dependences that 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

```
for(i=1...num_items) {                          } Each iteration is an epoch
  row = stock_table.select(items[i]);           }
  row.quantity--;                               }
  stock_table.update(items[i], row);            }
}


Row
StockTable::select(ItemId item)
{
  num_selects++;
  return *btree_lookup(item);
}

void
StockTable::update(ItemId item, Row row)
{
  num_updates++;
  Row *bt_row = btree_lookup(item);
  *bt_row = row;
}
```

Fig. 4. Simplified main loop from the NEW ORDER transaction.

transactions running on a full-featured DBMS, and by describing this process in detail for several TPC-C transactions.

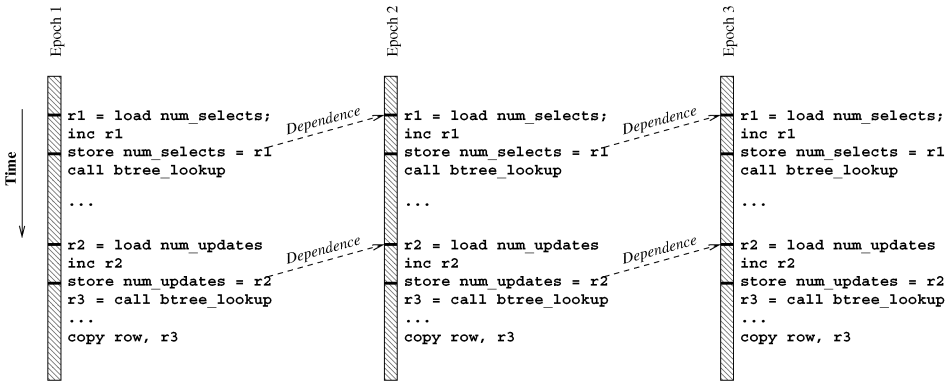
2. APPLYING TLS TO DATABASE TRANSACTIONS

In this section we provide an overview of the process of using TLS to incrementally parallelize a database transaction. We begin by discussing the issues involved in dividing a transaction into epochs. We then present the details of managing epochs and the underlying speculative threads. Finally, we introduce several techniques and tools for tolerating data dependence violations between epochs, and optimizing the performance of large epochs.

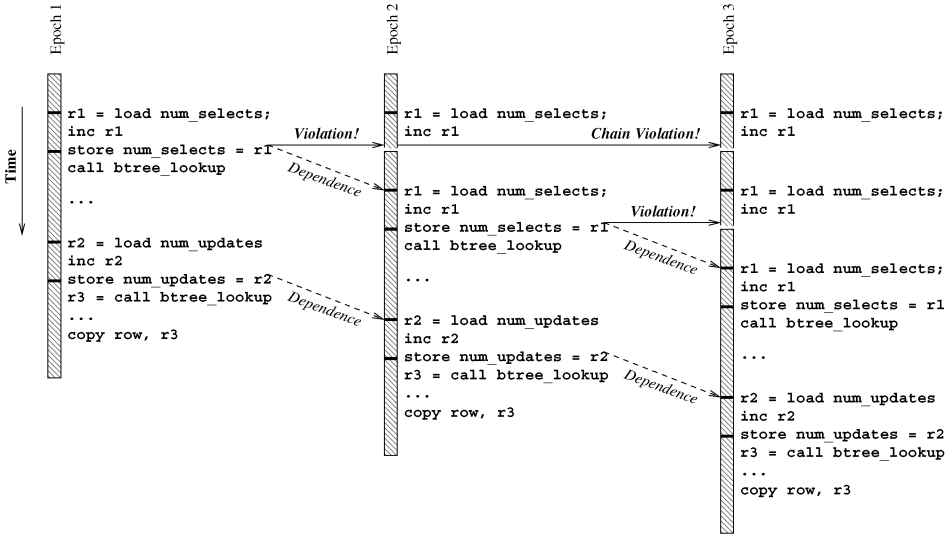
2.1 Dividing a Transaction into Epochs

Consider the `for`-loop in Figure 4, which is a highly simplified version of the central loop of the NEW ORDER transaction from TPC-C. This loop simply finds items in a database table and decrements their `quantity` field. Let's assume that the programmer knows that the elements in the `items` array are disjoint—this means that if the programmer looked at just the loop, they would presume that the loop is parallel, and each loop iteration could be run as a parallel *epoch*. If a transaction programmer were trying to apply TLS, then this loop would be a good candidate for parallelization.

Figure 5 illustrates the actual speculative execution of the example loop from Figure 4. As shown in Figure 5(a), the `select` and `update` functions increment the variables `num_selects` and `num_updates`, causing *data dependences* between our epochs. Furthermore, the `select` and `update` functions call `btree_lookup`, and we do not know what dependences may exist in the B-tree code.



(a) Execution of epochs in parallel results in dependences backward in time.



(b) Backward dependences cause misspeculation. Violations recover from misspeculation by restarting the epoch.

Fig. 5. TLS execution of the example loop from Figure 4.

The data dependences between epochs must be preserved for correct execution. To preserve dependences any load in epoch 2 must load the correct value from the *last* store to that memory location by epoch 1 or any earlier epoch. In Figure 5(a) we show several *backward dependences*, where the load executes before the last store in the previous epoch executes. When TLS detects a backward dependence, it is known as a *dependence violation*, and it restarts the epoch (Figure 5(b)).[2] In addition to restarting the epoch, all later epochs

[2]In this discussion we assume that only backward dependences can cause violations, because the TLS hardware [Colohan et al. 2006] makes an epoch's stores (updates) available to later epochs as soon as possible. Using such a design with aggressive update propagation ensures that the number of violations is minimized, which is important when the epochs are large (and hence the penalty of a violation is large).

are restarted as well through a *chain violation* since they may have consumed incorrect speculative values from the violated epoch. When the violated epoch reexecutes, the backward dependence that triggered the violation is turned into a forward dependence, and hence the correct result is computed.

Dependence violations cause work to be discarded and reexecuted, and hence they limit performance. The following techniques have been proposed in previous work to mitigate the performance impact of violated data dependences.

—*Limit epoch sizes.* Restarting large epochs throws away a large amount of work, which is inefficient. Using small epochs limits the amount of wasted work when a violation occurs.

—*Choose epochs to avoid dependences.* Dependences can be avoided by carefully choosing when to apply TLS at all, and by carefully choosing epoch boundaries [Vijaykumar 1998].

—*Choose epochs to avoid* backward *dependences.* Sometimes dependences can not be avoided completely—in these cases it makes sense to either choose epochs to try to avoid backward dependences, or to apply compiler scheduling techniques to the epochs in an attempt to turn backward dependences into forward dependences [Vijaykumar 1998; Zhai et al. 2004].

—*Use compiler managed synchronization.* If a dependence occurs frequently, insert explicit synchronization between the last store in an epoch and the first load of the next epoch to avoid a violation [Zhai et al. 2004].

—*Use hardware managed synchronization.* If hardware detects a load-store pair that causes frequent violations, insert synchronization ensuring that the load does not issue until the store retires [Steffan et al. 2002].

—*Use value prediction.* Detect which loads frequently cause violations, and use a value predictor to predict the correct value consumed by the load [Oplinger et al. 1999].

To parallelize database transactions while minimizing transaction programmer effort, we started with a simple division into epochs: we made each loop iteration in the transaction into an epoch. The resulting epochs had the following properties:

—The epochs were large. Previous work has studied epochs with various size ranges, including 3.9–957.8 dynamic instructions [Vijaykumar 1998], 140–7735 dynamic instructions [Prabhu and Olukotun 2003], 30.8–2252.7 dynamic instructions [Steffan et al. 2002], and up to 3900–103,300 dynamic instructions [Garzarán et al. 2003]. The epochs studied in this article are quite large, with 7574–489,877 dynamic instructions.

—There were many violation-causing dependences between epochs. This makes intuitive sense given the significantly increased sizes of the epochs.

—The epochs were not amenable to compiler based synchronization techniques. The majority of dependences were on shared state managed by the DBMS code, and not the transaction code. The DBMS code is shared by many transactions, and synchronization inserted for the benefit of one transaction was not useful for the execution of other transactions.

—The execution challenged hardware-based prediction and synchronization techniques: a given epoch in our database transactions has tens to hundreds of potential data dependences that have to be correctly predicted, while any misprediction results in the entire epoch being rewound. Our most accurate hardware-based predictors were unable to predict and synchronize our transactions effectively.

To overcome these challenges we adopted the following two-pronged approach.

(1) We incrementally modified the DBMS to avoid or remove data dependences that frequently trigger violations. In this section we examine how removing or avoiding dependences affects performance; in Section 4 we apply these techniques to the DBMS to remove dependences.
(2) We extended hardware support to divide each epoch into *subepochs* (a.k.a. subthreads), so that a single violation does *not* cause the entire epoch to be rewound. We give a brief overview of the operation of sub-epochs in this section; more detail is available in previous publications [Colohan 2005; Colohan et al. 2006].

2.2 Life Cycle of an Epoch

To parallelize the loop from Figure 4, the transaction programmer needs to break the loop up into epochs. The code that does this is shown in Figure 6. Note that the loop body is untouched, and we have just inserted template code by hand (i.e., as the transaction programmer) to exploit TLS functionality. In particular, the inserted template code in Figure 6 does the following:

① A new function, `tfork`, is used to create a thread to run each loop iteration as an epoch.[3] Note that the loop is structured so that `tfork` can fail if there are no CPUs (or epoch contexts on a CPU) available to run another thread—this allows TLS to dynamically adapt to the number of available CPUs.
② Each thread receives arguments using a designated portion of the stack known as the *forwarding frame* [Steffan et al. 2005].
③ The boundaries of speculative execution are marked with the `become_speculative` and `become_nonspeculative` functions.
④ Template code ensures that the epochs commit their speculative changes in the original sequential program order by passing a *homefree token*—indicating that the epoch can no longer be violated by prior epochs—from one thread to the next. When a thread possesses the homefree token it is said to be *homefree* and can no longer be violated by an older epoch.

The execution of our sample loop with these primitives added is illustrated in Figure 7(a).

[3]Although we are showing new function calls that implement TLS functionality, in our implementation each of these function calls expands to be a single inline assembly instruction.

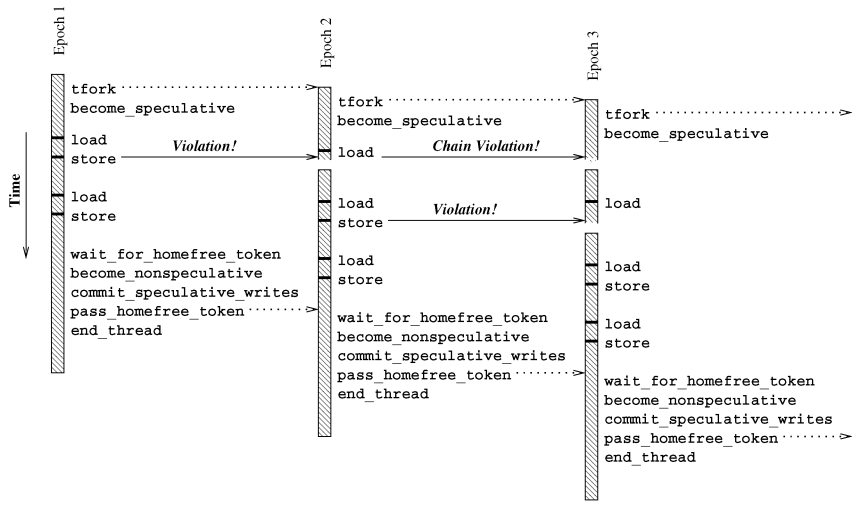
```
/* Structure for passing arguments                          }
 * to epochs: */                                            }
struct {                                                    }
   int i;                                                   } ②
} forward;                                                  }
forwarding_frame(&forward);                                 }
forwarding_size(sizeof(forward));                           }

for(forward.i=1...num_items) {
   /* Spawn thread to run next epoch: */
   ThreadDescriptor td = tfork();                           } ①
   if(td != 0) {
      /* Parent thread--child will
       * execute next loop iteration */
      int i = forward.i;                                    } ②
      become_speculative();                                 } ③

      row = stock_table.select(items[i]);                   }
      row.quantity--;                                       } Original loop body
      stock_table.update(items[i], row);                    }

      wait_for_homefree_token();                            } ④
      become_nonspeculative();                              } ③
      commit_speculative_writes();
      if(td != TFORK_FAILED) {
         pass_homefree_token(td);                           } ④
         end_thread();
      }
   }
}
```

Fig. 6. Example loop with TLS primitives.

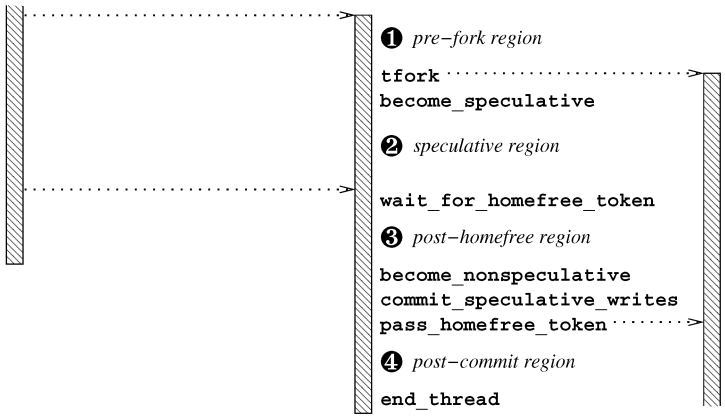
2.3 Moving Code to Avoid Dependences

By making these basic TLS primitives (forking epochs, homefree token passing, speculation boundaries) visible to software, we allow the software to be flexible in its use of epochs. In Figure 7(b), we show four interesting regions of an epoch's execution where transaction code can be placed. The default location for all of the code in the epoch is in the *speculative region* (❷ in Figure 7(b))—code placed in the speculative region executes speculatively, and the TLS mechanism ensures that execution in the speculative region is equivalent to the original sequential execution. When parallelizing software using TLS, the programmer would first place all code in the speculative region. Profile feedback will show if a data dependence is causing frequent violations. The programmer can then attempt to move the offending code upwards or downward into the other regions, as described below.

Most loops parallelized with TLS contain a loop index computation. The loop index computation can be as simple as incrementing an integer, or can



(a) TLS primitives in action.



(b) Regions of an epoch's execution.

Fig. 7. Detailed view of an epoch showing TLS primitives and how they break an epoch into regions of execution.

involve a linked list traversal or moving a database cursor. The loop index computation frequently causes a data dependence between epochs, but the index computation is usually not dependent on the body of the loop. If the loop index computation has no side effects then it can be moved up above the speculative region of the loop to the *prefork region* where the loop index is computed before starting the following epoch (❶ in Figure 7(b)). The `tfork` call effectively acts like a synchronization primitive between epochs, ensuring that the loop index computation for an epoch completes before the next epoch begins. This avoids violations due to the loop index. If the `tfork` primitive supports argument passing between threads (in this article we assume it does) then the loop index value becomes an argument to the next epoch.

Any code placed in the *post-homefree region* (the portion of an epoch's execution after the homefree token has arrived but before the token is passed on to the next epoch) (❸ in Figure 7(b)) will not be violated by an earlier thread, since waiting for the homefree token ensures that earlier threads will have committed all of their speculative writes. If some code frequently causes violations, and if that code's execution can be delayed without affecting correctness, then delaying it until the homefree token has arrived (the posthomefree region) will avoid violations. An example of this is the generation of log sequence numbers in a database system: an epoch in a transaction only generates log sequence numbers, and never consumes them. Because no code in the epoch depends on the generated numbers, it is safe to delay their generation until the posthomefree region.

If too much code is relocated to the posthomefree region then the homefree token may become a bottleneck, serializing execution. To avoid this, one can move code further down, after speculation has committed and the homefree token has been passed to the next epoch. This region of execution is known as the *postcommit region* (❹ in Figure 7(b)). To delay execution until after the homefree token has been passed the code must be thread safe, since it will be executed nonspeculatively in parallel with other threads. An example of code that can be delayed until the postcommit region is any call to `free`. When a transaction frees memory, its execution will be unchanged if the `free` is delayed until after the epoch commits.

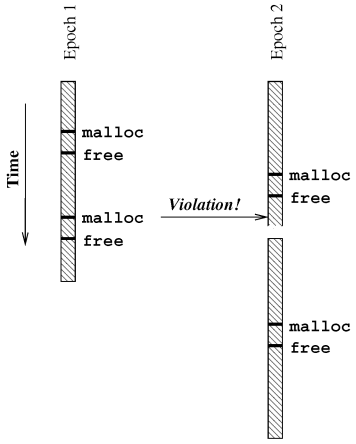
In Section 3 we will show numerous examples of how moving code out of the speculative region and into the prefork region, posthomefree region, and postcommit region can avoid violations when parallelizing the DBMS.

2.4 Avoiding Dependences by Escaping Speculation

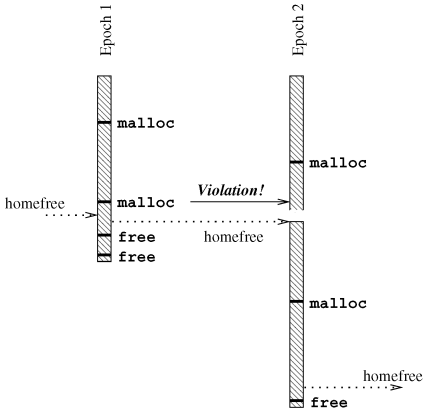
Dependences between epochs can often be easier to understand if you look at the higher-level operations being performed, instead of focusing on the individual loads and stores that cause the dependence. For example, consider two epochs that invoke `malloc` and `free`, as shown in Figure 8(a). Both the `malloc` and `free` routines read and modify shared data structures, namely, the free list maintained by the memory allocator. Because of this, any invocations of `malloc` or `free` that occur out of the original sequential program order will cause violations.

Since we know that the system allocator is thread safe, it is safe to invoke in the postcommit region. We observe that delaying the freeing of some memory will not affect the correct execution of the program. Therefore it is safe to avoid any dependences caused by `free` by moving the call to `free` down to the postcommit region (Figure 8(b)). A similar approach is used by Shinnar et al. [2004] for restoring program state when handling exceptions.

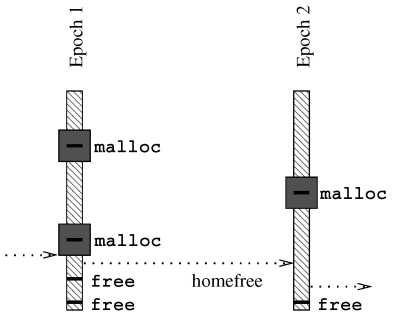
It is not possible to move the call to `malloc` downward, since the epoch can not proceed until the requested memory is allocated. Instead, we avoid the dependence by *escaping* the speculation mechanism. Fundamentally, to escape speculation we nonspeculatively allocate the memory when requested, and *recover* by freeing the memory again if speculation fails. To escape the



(a) Dependence between `malloc` and `free` causes a violation



(b) Delaying `free` until after homefree dependence between `malloc` and `free`, but exposes dependence between `malloc` and `malloc`.



(c) Escaping speculation for `malloc` eliminates last dependence.

Fig. 8. Removing dependences between `malloc` and `free`.

speculation mechanism, we wrap the `malloc` function with a routine that temporarily disables the speculation mechanism while executing `malloc`. This wrapper must also carefully check the arguments to `malloc` (to avoid overly large or frequent memory allocations, that could cause memory exhaustion), and register a handler that will call `free` on the allocated memory if the epoch is later violated.

The code wrapper shown in Figure 9 implements this modified version of `malloc`, as might be done by the DBMS programmer. In particular, this code does the following:

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

② Acquires a mutex that provides mutual exclusion between epochs within a transaction, to guard against the possibility that `malloc` was not implemented with intra-transaction concurrency in mind. Note that most implementations of `malloc` are indeed thread safe, so this extra paranoia can be eliminated once the programmer confirms this.

```
void *malloc_wrapper(size_t size) {
   static intra_transaction_mutex mut;     }②
   void *ret;

   suspend_speculation();                  }③
   check_malloc_arguments(id);             }①
   acquire_mutex(&mut);                    }②

   ret = malloc(size);

   release_mutex(&mut);                    }②
   on_violation_call(free, ret);           }④
   resume_speculation();                   }③

   return ret;
}
```

Fig. 9. Wrapper for the `malloc` function that escapes speculation to avoid dependences.

③ Temporarily escapes speculation. While speculation is escaped, the epoch is *nonspeculative* 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). Since no speculative reads are performed, the reads performed by `malloc` will not cause violations.

④ Saves a pointer to the recovery function, `free`. If the epoch is violated then `free` will be called to undo the memory allocation. This is similar to nested top actions in ARIES [Mohan et al. 1992], since we modify the execution but preserve higher-level semantics.

Escaping speculation simplifies coding: instead of redesigning the memory allocator to be amenable to TLS execution, we place this simple wrapper around the allocation function. However, this method requires that the `malloc` function be an *isolated undoable operation*. The `malloc` function is undoable: calling `free` undoes the call to `malloc`. The `malloc` function is also isolated: when it is undone via `free` no other transaction or earlier epoch is forced to rewind or otherwise alter its execution. We can apply the technique of escaping speculation to any operation that satisfies the isolated and undoable properties—in Section 4 we will see further examples of escaping speculation in use.

It is important to note that since we may be allocating resources speculatively, we must carefully handle error cases such resource exhaustion. For example, a speculative thread may attempt to `malloc` an amount of memory based on a large and incorrect value. In such cases and others that trigger significant system events, we want to abort all speculation—that in this case will free all memory speculatively allocated. This will not repair the resulting heap fragmentation, but will prevent the system from suffering from significant negative events caused solely by speculation.

2.5 Subepochs

Data dependences between epochs cause failed speculation, limiting performance. If the transaction was executing on a dataflow architecture [Arvind

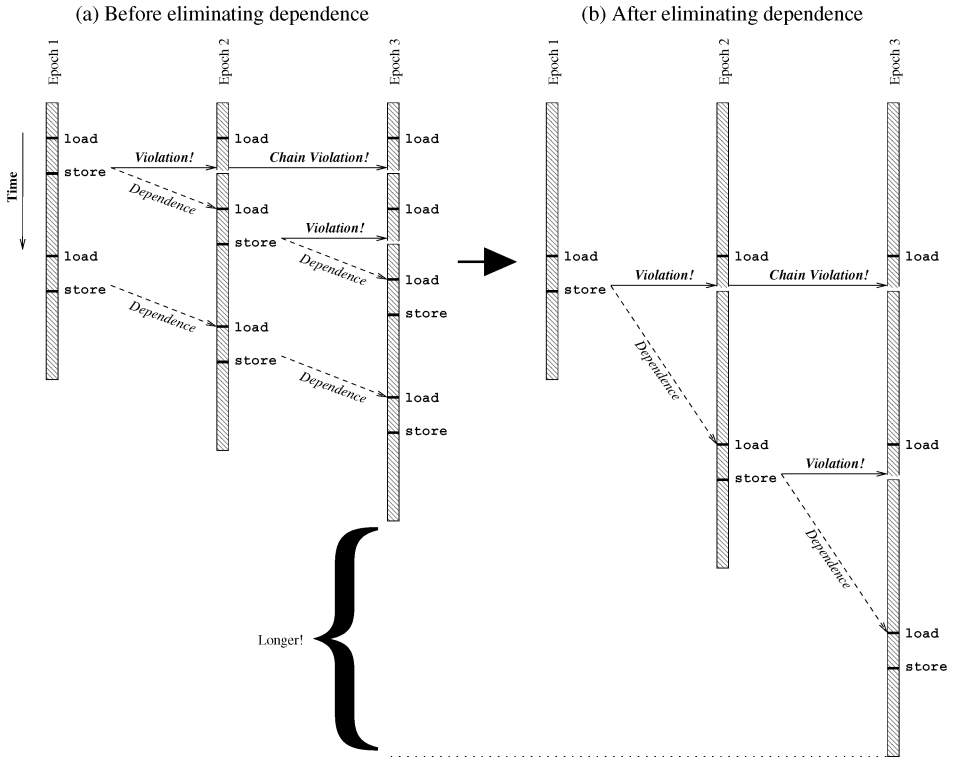


Fig. 10. Eliminating the first dependence in an epoch can *hurt* performance.

and Culler 1986] then modifying or rearranging the code to remove data dependences would directly lead to a performance improvement. Unfortunately, under TLS-style execution performance is limited not only by dependences, but also by *where* in the epoch they are located. A dependence located early in an epoch causes only a small amount of execution to be rewound, while a dependence located late in an epoch causes most of the epoch to be rewound. In Figure 10 we see that if a programmer eliminates a dependence early in the epoch's execution then it may *hurt* performance by exposing a dependence that occurs later in the epoch's execution.

Performance is hurt because on a violation the TLS mechanism rewinds both the misspeculated execution that depends on the errant load *and* all of the correct execution that precedes the errant load. When a dependence occurs late in the epoch's execution then more correct execution is rewound. We can limit the amount of correct execution rewound by using *subepochs*. A subepoch can be viewed as a *checkpoint* of an epoch. Each epoch is divided into multiple subepochs, and the state of each subepoch is maintained by the hardware until the entire epoch commits. If a violation is detected, it detects which subepoch contained the dependent load, and only that subepoch (and later subepochs) is restarted. Further detail on subepochs is available in a previous publication [Colohan et al. 2006, 2007], although we provide a brief summary here.

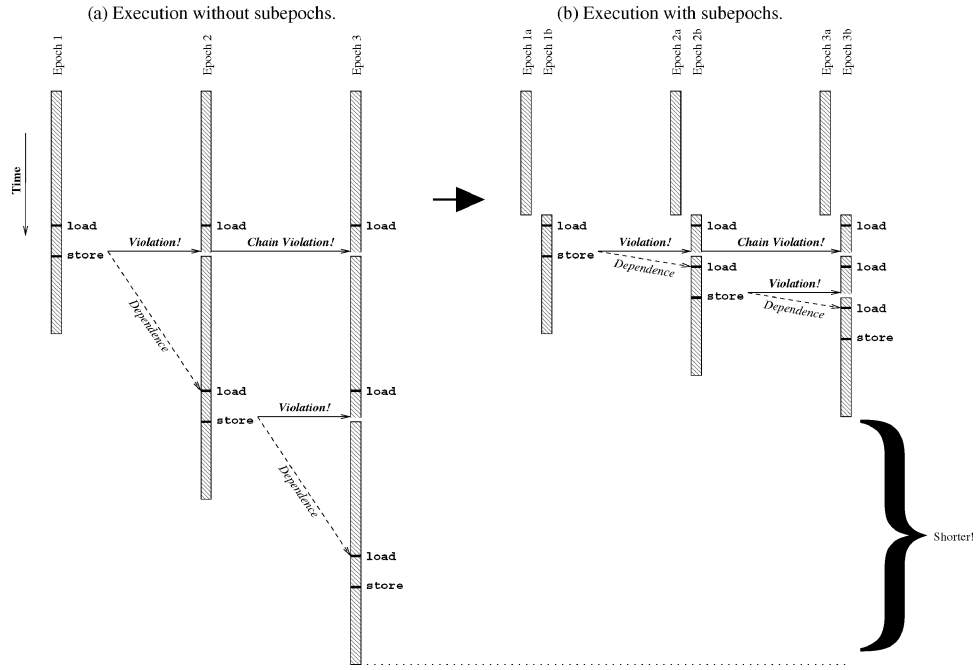


Fig. 11. Subepochs reduce the impact of a dependence late in the epoch.

Figure 11 shows the effect of dividing each epoch into two subepochs. Since the errant load occurs shortly after the second subepoch starts, very little correct execution is rewound when the violation is detected. If you compare Figures 10(a) and 11(b) you see that, when using subepochs, eliminating the first dependence in Figure 10(a) *improves* performance.

If using subepochs had no cost, then the best performance would be obtained by starting a new subepoch before every load instruction—this would completely avoid rewinding correct execution when a violation occurs. We have demonstrated that each subepoch consumes finite hardware resources [Colohan et al. 2006, 2007]: each additional subepoch adds state to each cache line and adds complexity to dependence tracking logic. To conserve these resources, we adopt a scheme where new subepochs are started periodically during the execution of an epoch (every n instructions issued), so that the maximum number of correct instructions discarded due to a violation is less than n.

2.6 Intertransaction Data Dependences

Up until this point we have discussed the TLS execution of a single transaction. A transaction runs on a *thread*[4] that may be further subdivided into epochs. How do the speculative epochs within a transaction's thread interact with other threads in the DBMS?

[4]In this article a *thread* refers to the unit of parallelism used by software without TLS. A database system typically maintains a pool of threads, and each thread runs a transaction. TLS adds parallelism by further dividing each thread into epochs.

As a thread executes, it performs loads and stores to memory, and acquires and releases mutexes and locks. All other threads in the system interact with the thread by observing the results of these memory and synchronization operations. To the other threads in the system, a thread that has been divided into epochs with TLS looks like any other thread, but with bursty store behavior—an executing epoch performs no externally visible stores, and all of the epoch's stores become visible when the epoch commits. You might imagine that performing the stores in a batch instead of in their original program order could introduce concurrency bugs, but modern parallel software already has to tolerate store reordering in hardware, and uses explicit synchronization for communication, based on release consistency [Gharachorloo et al. 1990].[5]

When an epoch executes, it speculatively executes all loads assuming that the loaded values will not change in memory before the epoch commits. If a loaded value is changed before the epoch commits then it triggers a violation, restarting the epoch from the start of the appropriate subepoch. The loaded value can be changed by three sources: (i) by a store performed by an earlier epoch in the same thread; (ii) by a store performed by any nonspeculative thread; or (iii) by an epoch from another thread committing. This means that stores from the other threads in the system can cause an epoch to be violated.

Does this mean that epochs will be constantly violated by the other threads (running other transactions) in the system? Not at all. A violation is caused by a *data dependence* between the other thread and the epoch. In Section 4 we show software transformations that eliminate data dependences between epochs in a single thread. These software transformations also serve to eliminate data dependences *between* threads.

2.7 Summary

We have demonstrated how TLS lets programmers introduce parallelism into a transaction without having to fully understand what data dependences exist in the DBMS's code. We showed that once a transaction has been speculatively parallelized, data dependences between epochs can be tolerated through code motion, escaping speculation, and subepochs. Finally, we demonstrated how a transaction's use of TLS does not impact the correctness of other transactions, although data dependences between other transactions and a transaction using TLS may incur additional violations. In the two sections that follow, we show how TPC-C transactions can be divided into epochs by a transaction programmer, and how the DBMS programmer can tune the DBMS to improve the resulting performance of the speculatively parallel transactions.

3. THE TRANSACTION PROGRAMMER: DIVIDING TRANSACTIONS INTO EPOCHS

To apply TLS to database transactions, the transaction programmer (or compiler) must first divide the transaction code into epochs. We applied TLS

[5]Section 3 contains further details on how synchronization primitives such as mutexes and locks are correctly handled in TLS execution.

to the transactions from the TPC-C benchmark [Gray 1993], representing an important class of commercial workloads. Since the TPC-C benchmark specification [Transaction Processing Performance Council 2005] describes the transactions in English, the first thing we required was an implementation. We implemented the five transactions from TPC-C (NEW ORDER, DELIVERY, STOCK LEVEL, PAYMENT and ORDER STATUS) on top of the BerkeleyDB storage manager [Olson et al. 1999]. The implementation is a straightforward and reasonably efficient implementation of the transactions as specified, and does not contain extensive performance optimizations.

The transaction programmer has limited knowledge of the internals of the DBMS, and chooses epochs to minimize data dependences that are apparent in the transaction code they are writing. In Section 4 we will show how to eliminate the most frequent dependences within the DBMS, so that the dependences in the transaction code are all that the transaction programmer has to consider. For the transactions in this article, the transaction programmer interacts with the DBMS through the following four basic operations on tables.

—*Select.* The select operation locates the specified row in the table and reads it.
—*Update.* The update operation locates the specified row in the table and modifies it.
—*Insert.* The insert operation adds a new row to the table.
—*Cursors.* A cursor is used to scan through a number of items in the table.

From the transaction programmer's perspective, dependences between epochs are caused by reads and writes performed by the transaction, and by the database operations performed by the transaction. Database operations can form dependences in two ways: (i) a read-after-write (RAW) dependence occurs when an epoch performs an update operation (or updates a row through a cursor), and a later epoch performs a select operation (or reads a row through a cursor); (ii) an insert dependence occurs when a later epoch performs a select operation (or reads a row through a cursor) then an earlier epoch inserts a new row that changes the result of the later epoch's select operation.

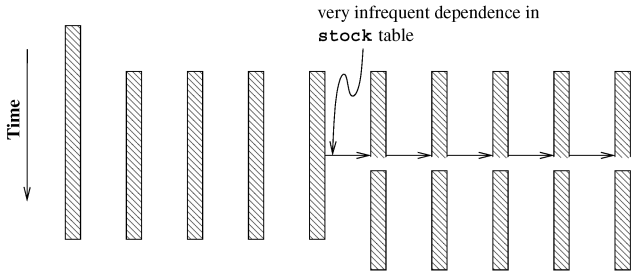
The TLS mechanism can detect when a RAW or insert dependence on the database data violates the original sequential order, and will restart the epoch or subepoch as appropriate. To the TLS mechanism database operations look just like any other memory operations—assuming as we do that data accessed by a transaction is *pinned* in memory for the duration of the transaction (i.e., not swapped out to disk). This is because the database data is mapped into memory whenever it is used. As a result, the TLS mechanism is able to detect violations caused by database operations using the same mechanism that lets it detect violations caused by memory operations.

In the following sections, we look at each transaction from TPC-C in detail, examining the dependences that occur due to reads and writes to local variables and due to database operations.

```
// New Order:  Customer ordering a list of items from a
// warehouse.

w_row = warehouse_table.select(w_id);
c_row = customer_table.select(w_id, d_id, c_id);

d_row = district_table.select(w_id, d_id);
o_id = d_row.next_o_id;
d_row.next_o_id++;
district_table.update(w_id, d_id, d_row);

o_row.id = c_id;
o_row.carrier = 0;
order_table.insert(w_id, d_id, o_id, o_row);
neworder_table.insert(w_id, d_id, o_id);

for(i=1...num_items) {
   i_row = item_table.select(items[i]);
   st_row = stock_table.select(items[i]);
   st_row.quantity--;
   stock_table.update(items[i], st_row);

   ol_row.item = items[i];
   ol_row.price = i_row.price;
   orderline_table.insert(w_id, d_id, o_id, i, ol_row);
}
```

Parallelize this loop

(a) Simplified transaction source code.



(b) Transaction programmer's expected execution with TLS.

Fig. 12. The NEW ORDER transaction.

3.1 New Order

In Figure 12(a) we show a simplified version of the main loop from NEW ORDER. The NEW ORDER transaction is the main transaction from the TPC-C workload, representing at least 45% of executed transactions. NEW ORDER executes on behalf of a customer placing an order for a list of 5–15 items from a warehouse.

We have chosen to make each loop iteration into an epoch. Since the loop covers 78% of the transaction's execution time, parallelizing it should result in a substantial performance benefit. Examining the code in Figure 12(a), it appears that the only dependence between epochs is due to reads and writes to the *stock* table. The benchmark specifies that the `items` to be purchased will

be chosen randomly from a uniform distribution of 100,000 items. This means that subsequent epochs are very unlikely to access the same item and cause a data dependence violation.

In Figure 12(b) we show the expected TLS execution: first, the code before the loop begins is executed. Then the loop is run in parallel, and the only violations are caused by infrequent data dependences in the *stock* table. Recall that when a violation occurs a chain violation causes all later epochs to restart since they may have consumed invalid results from the violated epoch. Since violations are infrequent, we expect this loop to perform quite well—in Section 5 we will see that this is true.

3.2 Delivery

The DELIVERY transaction (Figure 13(a)) loops through all of the districts in a warehouse and delivers the oldest outstanding order in each district. This transaction presents two possibilities for parallelization—the inner and outer loops. We chose to parallelize both the inner loop and the outer loop separately (we call the outer loop variant DELIVERY OUTER).

The inner loop represents 63% of the transaction's execution time. We parallelize this loop such that each iteration is an epoch. The only dependence between epochs is caused by the update of the variable `ol_total`. Although it is possible to use accumulator variable expansion[6] [Mahlke et al. 1992] to transform the transaction to avoid the `ol_total` dependence, we assume that the transaction programmer has not optimized it away. When executed, the dependence causes a violation at the end of each and every epoch, as shown in Figure 13(b). From this figure one can see that a frequent dependence violation that causes a small amount of execution to be rewound will have a small impact if TLS is using a small number of CPUs. As the number of CPUs grows, the fraction of execution time spent reexecuting epochs grows, since the dependence becomes more and more of a bottleneck. In Section 5 we shall see that with up to 8 CPUs this is not a critical bottleneck.

The outer loop of the DELIVERY transaction has no dependences, as illustrated in Figure 13(c). In fact, it is completely parallel. The TPC-C benchmark specification allows implementors of TPC-C to take advantage of this by running each outer loop iteration as a separate transaction. We try parallelizing the outer loop using TLS instead of using separate transactions to explore what happens if TLS is used with very large epochs—the epochs in this outer loop decomposition contain an average of 490,000 instructions each.

3.3 Stock Level

The STOCK LEVEL transaction (Figure 14) is a read-only transaction that checks recently ordered items to see if any items in the warehouse have almost run out. This transaction contains one loop that dominates the transaction's execution,

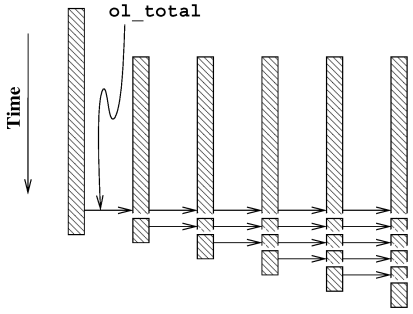
[6]Accumulator variable expansion is used to eliminate a data dependence caused by a variable that accumulates a value, such as a sum in a dot product. In the DELIVERY example each epoch would be given a private copy of the `ol_total` variable to update, and once the loop was complete all of those private variables would be summed to generate the final value.

```
// Delivery:  deliver the oldest order in each district in
// the warehouse.

for(d_id=1...DIST_PER_WAREHOUSE) {
   cursor = new_order_table.new_cursor(w_id, d_id,
                                       OLDEST_O_ID);
   no_row = cursor.fetch_next();
   cursor.delete();
   no_o_id = no_row.id;

   o_row = order_table.select(no_o_id, w_id, d_id);
   o_row.carrier_id = carrier_id;
   order_table.update(no_o_id, w_id, d_id, o_row);

   ol_total = 0;

   for(item=1...o_row.ol_cnt) {
      ol_row = orderline_table.select(w_id, d_id,
                                      no_o_id, item);
      ol_row.date = date();
      ol_total += ol_row.amount;
   }

   c_row = customer_table.select(c_id);
   c_row.balance += ol_total;
   order_table.update(c_id, c_row);
}
```

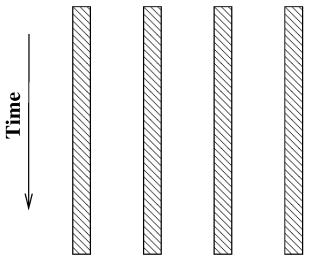
Parallelize this loop

Parallelize this loop

(a) Simplified transaction source code.



(b) Transaction programmer's expected execution with TLS.



(c) Transaction programmer's expected execution with TLS—outer loop.

Fig. 13. The DELIVERY transaction.

representing 98% of the transaction execution time. We can parallelize the main loop so that each iteration is an epoch.

The main loop of STOCK LEVEL iterates over a table using a *cursor*[7]—at the start of each epoch the cursor is read, and at the end of each epoch the cursor is

[7]A cursor is a transaction visible pointer to a location in a database table. A cursor is typically used for traversing the table to scan a range of records in the table. Database systems can also use cursors internally for implementing B-tree searches.

```
// Stock Level:  examine all items ordered in the last 20
// orders to see if stock is running low.

low_stock = 0;
d_row = district_table.select(w_id, d_id);
found_items = empty_set();

cursor = orderline_table.new_cursor(w_id, d_id,
                                    o_id - 20);
do {                                                        }
   ol_row = cursor.data();                                  }
   item_id = ol_row.id;                                     }
   if(!found_items.contains(item_id)) {                     }
      found_items.insert(item_id);                          }
                                                            }
      s_row = stock_table.select(w_id, item_id);            }
      if(s_row.quantity < threshold) {                      }
         low_stock++;                                       }
      }                                                     }
   }                                                        }
} while(cursor = cursor.next());                            }
```

Parallelize this loop

Fig. 14. Simplified transaction source code for the STOCK LEVEL transaction.

incremented. This forms a dependence from the end of each epoch to the start of the next epoch, that completely serializes execution. To avoid this problem, we turn the `do-while` loop into a `while` loop, as shown in Figure 15(a). This makes the critical path caused by the cursor as short as possible.

The cursor still forms a dependence between the epochs, that causes each epoch to be violated near the start of its execution, as shown in Figure 15(b). We could add explicit synchronization to the loop to avoid this violation, but we do not do so since we wish to demonstrate the performance gains possible with minimal effort by the transaction programmer. The violations caused by the cursor dependence causes the execution of the rest of the epochs to be somewhat *skewed* in time. This skew ensures that the infrequent dependences on the `found_items` set and `low_stock` variable that occur later in the epoch are executed *in-order*, preventing them from causing additional violations.

3.4 Payment

The PAYMENT transaction (Figure 16(a)) is a short transaction that records a payment made by a customer. This transaction contains no loops that cover a significant fraction of execution time. Since the last two operations on the DBMS are independent (updating the `customer` table and inserting into the `history` table), we run them as two parallel epochs, as shown in Figure 16(b). These two epochs cover only 30% of the transaction's execution, meaning that they should not offer a large performance gain.

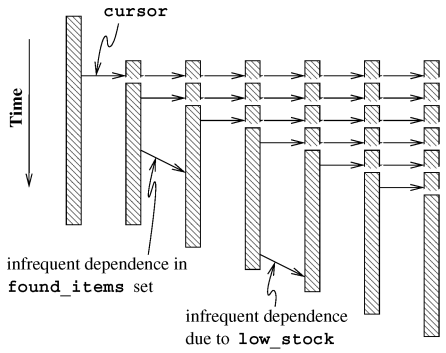
```
ol_row = cursor.data();
item_id = ol_row.id;
if(!found_items.contains(item_id)) {
   found_items.insert(item_id);

   s_row = stock_table.select(w_id, item_id);
   if(s_row.quantity < threshold) {
      low_stock++;
   }
}
```

Epoch 1

```
while(cursor = cursor.next()) {
   ol_row = cursor.data();
   item_id = ol_row.id;
   if(!found_items.contains(item_id)) {
      found_items.insert(item_id);

      s_row = stock_table.select(w_id, item_id);
      if(s_row.quantity < threshold) {
         low_stock++;
      }
   }
}
```

Epochs 2...n

(a) Do-while loop transformed into while loop to minimize impact of `cursor` dependence.



(b) Transaction programmer's expected execution with TLS.

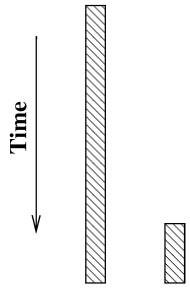
Fig. 15. Transformed code and expected execution for the STOCK LEVEL transaction.

3.5 Order Status

The ORDER STATUS transaction (Figure 17(a)) looks up the status of each item ordered by a customer. This transaction contains two loops that we parallelize that cover 38% of the transaction's execution. The majority of the work done in each loop iteration is a cursor lookup and increment, forming a dependent chain. Both of these loops are dominated by a dependence on the cursor used in them, so as we will find later when executed speculatively in parallel the execution is mostly serialized (shown in Figure 17(b)).

```
// Payment:  Record customer's payment.

w_row = warehouse_table.select(w_id);
w_row.ytd += payment_amount;
warehouse_table.update(w_id, w_row);

d_row = district_table.select(w_id, d_id);
d_row.ytd += payment_amount;
district_table.update(w_id, d_id, d_row);

if(byname) {
    // Customer specified by name.  Find all of the
    // customers who's name matches, and pick the one
    // in the middle:
    namecnt = count_rows(customer_table.select(customer_name));
    cursor = customer_table.new_cursor(customer_name);
    for(i=1...namecnt/2) {
        cursor = cursor.next();
    }
    c_row = cursor.data();
} else {
    c_row = customer_table.select(c_id);
}
c_row.balance += payment_amount;                    } Thread 1
customer_table.update(c_id, c_row);                 }
h_row.date = date();                                }
h_row.amount = payment_amount;                      } Thread 2
history_table.insert(w_id, d_id, c_id, h_row);      }
```

(a) Simplified transaction source code.



(b) Transaction programmer's expected execution with TLS.

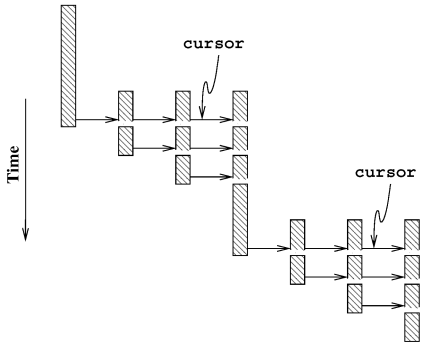
Fig. 16. The PAYMENT transaction.

3.6 Summary

We have summarized in detail the dependences that occur within TPC-C transactions due to local variables and database operations, and reasoned about how to go about dividing each transaction into epochs. We conclude that TLS parallelization is promising for three of the five TPC-C transactions. In the next section we discuss how the DBMS programmer can optimize performance by eliminating dependences between epochs.

```
// Order Status:  Find the last order by the customer and
// return the status of each item in the order.

if(byname) {
    // Customer specified by name.  Find all of the
    // customers who's name matches, and pick the one
    // in the middle:
    namecnt = count_rows(customer_table.select(customer_name));
    cursor = customer_table.new_cursor(customer_name);
    for(i=1...namecnt/2) {
        cursor = cursor.next();
    }
    c_row = cursor.data();
    c_id = c_row.id;
} else {
    c_row = customer_table.select(c_id);
}

cursor = order_table.new_cursor(c_id)
do {                                                    } Parallelize
    o_row = cursor.data();                              }   this loop
    o_id = o_row.id;                                    }
} while(cursor = cursor.next());                        }
i = 0;
cursor = order_line_table.new_cursor(o_id);

do {                                                    } Parallelize
    results[i++] = cursor.data();                       }   this loop
} while(cursor = cursor.next());                        }
```

(a) Simplified transaction source code.



(b) Transaction programmer's expected execution with TLS.

Fig. 17. The ORDER STATUS transaction.

4. THE DBMS PROGRAMMER: ELIMINATING DEPENDENCES AND PERFORMANCE TUNING THE DBMS

While our transaction-level analysis concludes that TLS parallelization is promising for three of the five TPC-C transactions, the implementation details of query execution algorithms and access methods in the DBMS reveal more potentially performance-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 memory allocation operation (`malloc`) typically uses a single pool of memory, hence 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) *Escape speculation for isolated undoable operations (IUOs).* This mechanism was introduced in Section 2.4. 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 that maintain LRU information. We prefer to allow `pin_page` operations to complete in any order. An epoch can simply call `pin_page` with speculation escaped: 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.*

(3) *Postpone operations until the end of the epoch.* Techniques for postponing operations were introduced in Section 2.3. 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 (making 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 remainder of this section, we tour the database system's major components, and explain how the database system programmer can 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 the management of resources, including 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 that track unused resources are shared between epochs. In the discussion that follows, we examine each of these resources and develop strategies for executing them in parallel.

4.1.1 *Latches.* The database system uses latches[8] 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 only briefly—in Section 4.1.2 we discuss *locks*, offering 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 (these may be implemented as a test-and-set, load-linked/store-conditional, atomic increment, etc.). This read-test-write cycle can cause a data dependence violation between epochs (Figure 18(a)).

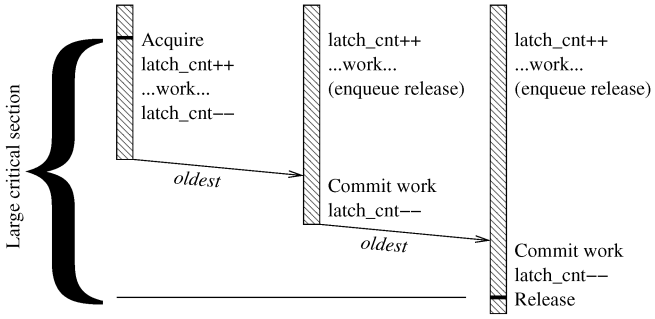
The TLS mechanism already ensures that any data protected by the latch is accessed in a serializable order *within a transaction*, namely, the original sequential program order. However, latches do ensure that mutual exclusion is maintained *between transactions*, and TLS does not perform that function. 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 the modifications made by an epoch are buffered until it commits, we must postpone all release operations until after the epoch has fully committed (the *post-commit region* of the epoch from Figure 7(b)). 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.

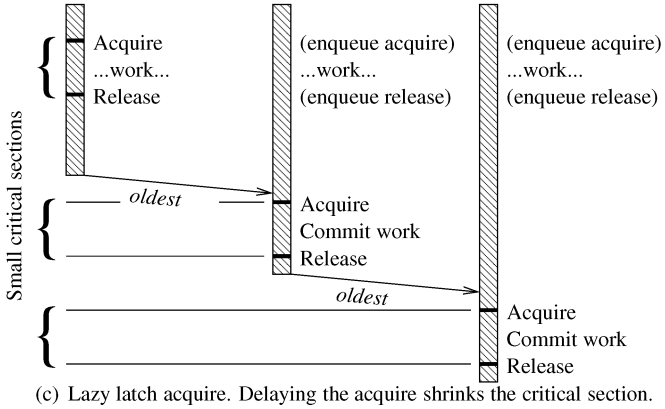
Next we consider acquire operations. During normal execution, when a latch is acquired it prevents other transactions in the system from changing the associated data. A naïve approach to handling a latch acquire under TLS is to perform the acquire *nonspeculatively* at the point when it is encountered. This can be implemented by a recursive latch that 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 18(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.

[8]The term *latch* is from the field of databases. A latch is equivalent to a *mutex* in operating systems parlance.

(a) Latch operations create dependences.



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



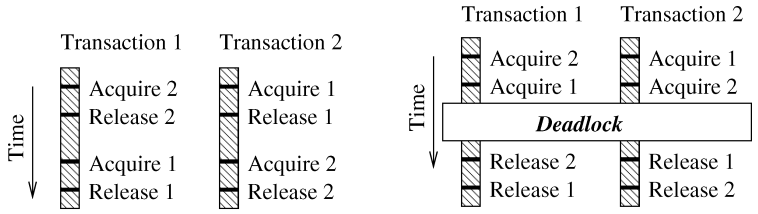
(c) Lazy latch acquire. Delaying the acquire shrinks the critical section.

Fig. 18. Adapting latches for use under TLS execution.

To avoid long critical sections, we can also postpone acquires to the *posthome-free* region of the epoch as shown in Figure 18(c). This *lazy* approach has three parts: (i) all latch acquires are performed at the end of the epoch, (ii) the buffered speculative modifications are committed, and finally (iii) all latch releases are performed. This 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.[9]

Both the lazy and aggressive latch schemes have a potential problem: they reorder the latch release operations relative to the latch acquire operations as specified in the original program. If multiple latches are acquired by a single

[9]This is similar to optimistic concurrency control [Kung and Robinson 1981], except that the optimism is at the granularity of an *epoch* instead of a *transaction*. A latch is held optimistically only for the duration of the epoch (instead of for the entire transaction), and when a conflict occurs only the epoch rewinds (instead of rewinding the entire transaction).



(a) Latch operations before reordering. (b) Latch operations after reordering.

Fig. 19. Delaying latch release operations until after a epoch commits can introduce deadlock.

epoch, a deadlock may emerge that is not possible in the sequential execution, as shown in Figure 19. 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 [Silberschatz et al. 2002], 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 timeout) by violating and restarting one of the deadlocked epochs. Forward progress is guaranteed because there is always at least one epoch (the oldest) that executes nonspeculatively. 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 *posthomefree* region of the epoch, since a handle must be returned. Instead, we return an *indirect* handle that 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 by recognizing that we can treat *read-only* and *read/write* locks differently: 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, and we then (ii) perform all *read/write* lock acquires that were encountered during the epoch, (iii) commit the epoch's changes to memory, and then (iv) perform all *read/write* 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.[10] If latches were labeled as read-only or read/write, then this optimization could also be applied to latches in addition to locks.

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 preallocated 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 that allocated it, increasing cache locality and eliminating dependences between epochs. An alternative technique would be to *escape* speculation when allocating cursors, as described in Section 2.4.

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 `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 [Berger et al. 2000] is one such allocator that maintains separate free lists for each CPU, so that most requests for memory do not communicate. An alternative technique would be to *escape* speculation when allocating memory, as described in Section 2.4.

To avoid dependences caused by `free` operations, we delay the execution of `free` operations until the *post-commit* region of the epoch.

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*.[11] This operation reads the requested

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

[11]The buffer pool is the subset of the database pages that currently reside in memory. The buffer pool manager (similar to a virtual memory manager in an operating system) is responsible for retrieving needed pages from disk and writing dirty pages back to disk in response to `pin_page` and `unpin_page` requests.

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 that is visible to the entire system. If the epoch that 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 [Garcia-Molina and Salem 1987].)

To execute the `pin_page` function nonspeculatively, we *escape speculation*, as described in Section 2.4. Relaxing ordering constraints simplifies coding: instead of redesigning the buffer pool to be amenable to TLS execution, we place a simple wrapper around the allocation function. 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 `malloc` are also isolated undoable operations, and so this code template could be applied to these functions instead of partitioning their free pools. The `lock_acquire` 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 lock and 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 in the *post commit* region of the epoch.

4.2 The Log

Every time the database is modified, the changes are appended to the log. For recovery to work properly (using ARIES [Mohan et al. 1992]), 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. In the *posthomefree* region of the epoch, 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 modifies 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. If such a violation happens frequently then it may be possible to change the B-tree comparison function so that subsequent inserts do not fall on the same leaf page.

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 many data dependences, but since they happen infrequently (by design), we can afford to just ignore them. In the rare cases when page splits occur, the TLS mechanism will ensure their correct sequential execution. The TLS mechanism provides a valuable fallback, 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 Statistics Gathering

The database system gathers statistics on its internal operations by incrementing counters. Every time a counter is incremented in two consecutive epochs, a dependence is created. Since these counters are frequently updated and rarely read, they are parallelized by creating a private copy of each counter per CPU. When the counter is read, the sum of all of the private values is computed.

4.5 Error Checks

This work indicates 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 that 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.

4.6 False Sharing

To minimize overhead, the TLS mechanism tracks data dependences at the granularity of a cache-line. However, accesses to different variables that happen to be allocated on the same cache line can cause data dependence violations due to false sharing [Torrellas et al. 1990; Eggers and Jeremiassen 1991]. This problem can be remedied by inserting padding to ensure that variables that are frequently accessed by different CPUs are not allocated on the same cache line [Jeremiassen and Eggers 1995].[12]

5. EXPERIMENTAL RESULTS

In this section we evaluate the incremental parallelization of transactions using TLS, and show the resulting performance gains.

5.1 Benchmark Infrastructure

Our experimental workload is composed of the five transactions from TPC-C (NEW ORDER, DELIVERY, STOCK LEVEL, PAYMENT and ORDER STATUS).[13] We have parallelized both the inner and outer loop of the DELIVERY transaction, and denote the outer loop variant as DELIVERY OUTER. We have also modified the input to the NEW ORDER transaction to simulate a larger order of between 50 and 150 items (instead of the default 5 to 15 items), and denote that variant as NEW ORDER 150. All transactions are built on top of BerkeleyDB 4.1.25. Evaluations of techniques to increase concurrency in database systems 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. A normal TPC-C run executes a concurrent mix of transactions and measures *throughput*[14]; since we are concerned with *latency*, we run the individual transactions one at a time. TLS improves the performance of CPU bound transactions: if a transaction spends the majority of its execution time awaiting buffer pool requests then the transaction is disk bound and not CPU bound. To ensure we are studying CPU bound transactions, we configure the DBMS with a large (100-MB) buffer pool.[15]

The parameters for each transaction are chosen according to the TPC-C run rules using the Unix `random` function, and each experiment uses the same seed for repeatability. The benchmark executes as follows: (i) start the DBMS;

[12]Insertion of padding works for most data structures, but is not appropriate for data structures that mirror disk-resident data, such as B-tree page headers. In this case, changes will have to be made to the B-tree data structure itself (see Section 4.3).

[13]Our workload was written to match the TPC-C spec as closely as possible, but has not been validated. The results we report in this article are speedup results from a simulator and not TPM-C results from an actual system. In addition, we omit the terminal I/O, query planning, and wait-time portions of the benchmark. Because of this, the performance numbers in this article should not be treated as actual TPM-C results, but instead should be treated as representative transactions.

[14]In a database context, throughput is the rate of transaction execution (transactions executed per minute).

[15]This is roughly the size of the entire dataset for a single warehouse.

Table I. Simulated Memory System 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 integers	1 cycle
FP divide	15 cycles
FP square root	20 cycles
All other FPs	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	10 cycles
Minimum miss latency to local memory	75 cycles
Main memory bandwidth	1 access per 20 cycles

(ii) execute 10 transactions to warm up the buffer pool; (iii) start timing; (iv) execute 100 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 that is linked with the benchmark that contains the transaction code.

To apply TLS to this benchmark, we started with the unaltered transaction, marked 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 that caused speculation to fail, 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 four-way issue, out-of-order, superscalar processors similar to the MIPS R14000 [Yeager 1996], 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 (GShare [McFarling 1993] with 16-kB, 8 history bits), instruction fetching, branching penalties, and the memory hierarchy (including bandwidth and contention) are all modeled, and are parameterized as shown in Table I. The TLS mechanism is implemented

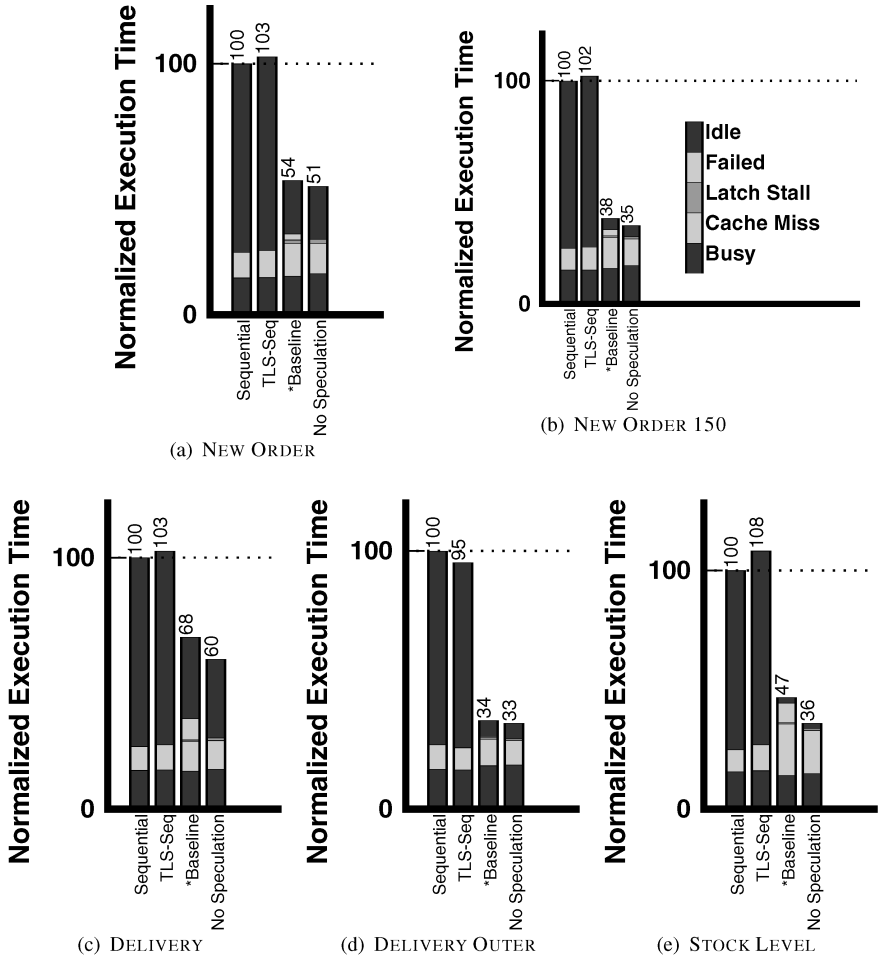
using the hardware design described in a previous publication [Colohan et al. 2006], including hardware support for large epochs and eight subepochs per epoch. Latencies due to disk accesses are not modeled, and hence these results are most readily applicable to situations where the database's working set fits into main memory.

The simulator used to generate these results is a trace driven simulator, meaning that the instruction stream of a sequential run is used to drive a parallel timing simulation. Trace driven simulation allows simulation results to be deterministic and allows us to model oracle hardware, but also means that accurately measuring interactions between parallel speculative threads is more challenging. For example, the wrong code paths due to misspeculation (both due to TLS and due to branch misspeculations) are simulated by executing the correct path twice, although such mispredicted paths are not permitted to prefetch instructions or data. Recovery code (such as code executed to undo misspeculated `page_get` or `malloc` calls) invoked on a violation is currently not simulated—these functions should take very little time to execute due to the small amount of work they must perform. When a violation is detected while speculation is escaped, the epoch restarts immediately, instead of delaying the violation until speculation resumes.

5.3 High-Level Benchmark Characterization

We start by characterizing the benchmarks, so we can better understand them, and measure them as shown in Figures 20 and 21. Each benchmark transaction is executed a number of different ways to illustrate the overheads of our approach and the potential for parallelism. As a starting point for comparison, we run our original sequential benchmark, showing the execution time with no TLS instructions or any other software transformations running on one CPU of the machine (configured with four CPUs, cache line replication, and subepoch support enabled). This SEQUENTIAL experiment takes between 17 and 509 million cycles (Table II) to execute, but this time is normalized to 100 in Figures 20 and 21 (note that the large percentage of *Idle* is caused by three of the four CPUs idling in a sequential execution). When we transform the software to support TLS, we introduce some software overheads that are due to new instructions used to manage epochs, and also due to the changes to the DBMS we made to parallelize it. The TLS-SEQ experiment in Figures 20 and 21 shows the performance of this parallelized executable running on a single CPU—the additional software overhead is reasonable, varying from −5% to 11% (negative overheads are due to our added code inadvertently improving the performance of the compiler optimizer).

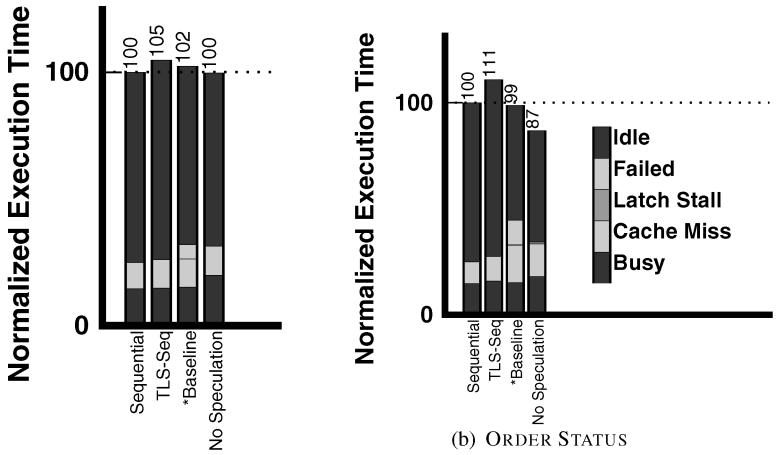
When we apply TLS with our BASELINE hardware configuration with four CPUs, eight subepochs per epoch, and 5000 speculative instructions per subepoch), we see a significant performance improvement for three of the five transactions, with a 46%–66% reduction in execution time. The PAYMENT and ORDER STATUS transactions do not benefit from TLS: as we saw in Section 3 the PAYMENT contains no loops worth parallelizing, and the threads we chose were limited by a dependence in the locking code that did not affect



Bar	Explanation
Sequential	No modifications or TLS code added.
TLS-Seq	Optimized for TLS, but run on a single CPU.
Baseline	Execution on hardware described in this article.
No speculation	Upper bound—modified hardware to treat all speculative writes as nonspeculative.

Fig. 20. Overall performance of optimized benchmark on four CPUs (remaining benchmarks shown in Figure 21).

the other transactions; the main loop in ORDER STATUS contains an unavoidable dependence on a cursor operation. As shown in Figures 20 and 21, the fraction of execution time wasted on failed speculation for the baseline experiment is small. However, it is important to understand that nearly 100% of epochs suffer at least one dependence violation. Hence these results demonstrate the overall effectiveness of subepochs in tolerating dependence violations.



(a) PAYMENT (b) ORDER STATUS

Bar	Explanation
Sequential	No modifications or TLS code added.
TLS-Seq	Optimized for TLS, but run on a single CPU.
Baseline	Execution on hardware described in this article.
No speculation	Upper bound—modified hardware to treat all speculative writes as nonspeculative.

Fig. 21. Overall performance of optimized benchmark on four CPUs.

Table II. Benchmark Statistics

Benchmark	Sequential Exec. Time (Mcycles)	Coverage	Average Epoch Stats		
			Size (Dyn. Instrs.)	Spec. Insts. Per Epoch	Threads per Transaction
NEW ORDER	62	78%	62k	35k	9.7
NEW ORDER 150	509	94%	61k	35k	99.6
DELIVERY	374	63%	33k	20k	10.0
DELIVERY OUTER	374	99%	490k	327k	10.0
STOCK LEVEL	253	98%	17k	10k	191.7
PAYMENT	26	30%	52k	32k	2.0
ORDER STATUS	17	38%	8k	4k	12.7

Is it possible to do better? In the NO SPECULATION experiment, we show the performance if the same program is run purely in parallel, incorrectly treating all speculative memory accesses as nonspeculative (and hence ignoring all data dependences between epochs)—this is an upper bound on performance, since it shows what would happen if speculation never failed and if no cache space was devoted to the storage of speculative state. This execution does not show linear speedup due to the nonparallelized portions of execution (Amdahl's law), and due to a loss of locality and communication costs due to spreading of execution over four caches. We find that for NEW ORDER, NEW ORDER 150, and DELIVERY OUTER, we are very close to this ideal, and further optimization is not worthwhile. For DELIVERY further improvements are limited by an output dependence in the ORDER LINE table. The STOCK LEVEL

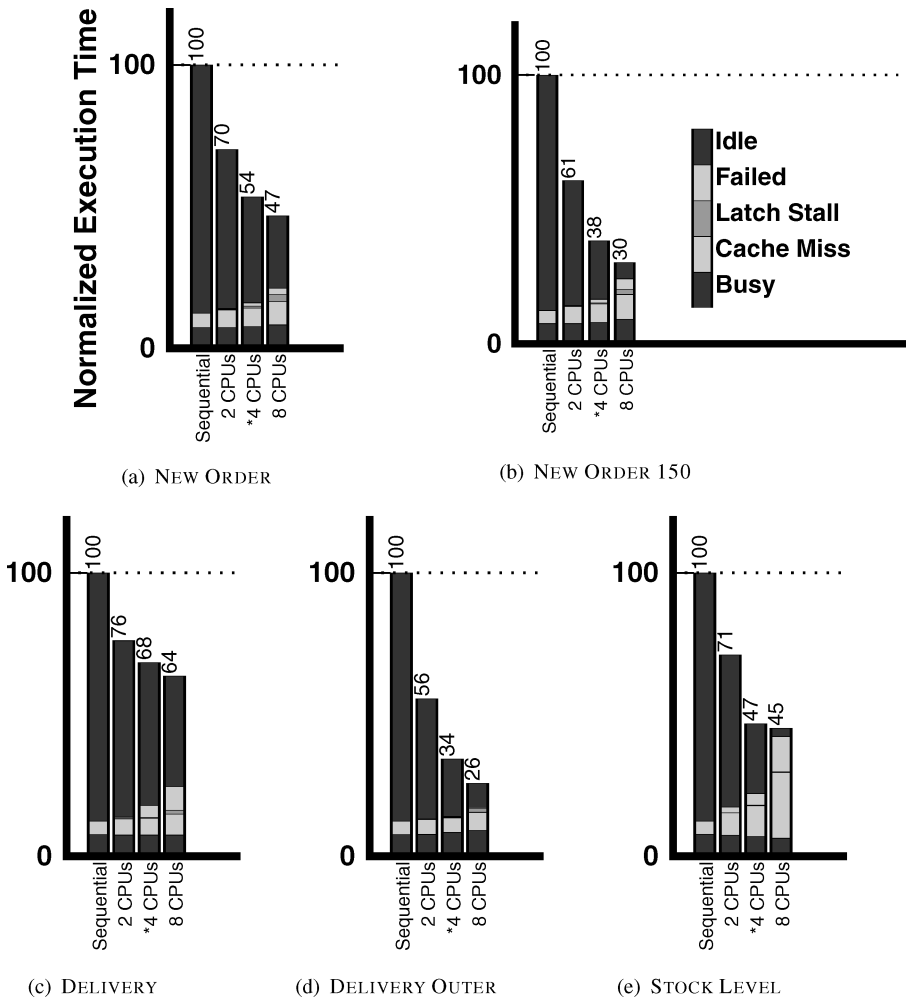
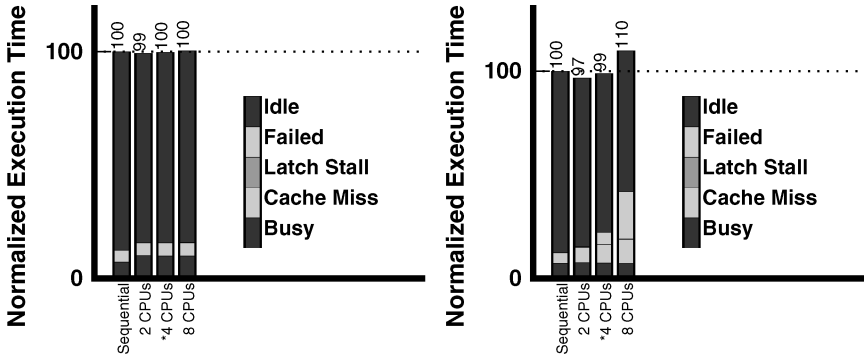


Fig. 22. Performance of optimized benchmark while varying the number of CPUs (remaining benchmarks in Figure 23).

transaction is limited by dependences on the cursor used to scan the ORDER LINE table.

5.4 Scaling Intratransaction Parallelism

In Figures 22 and 23 we see the performance of the fully optimized transactions as the number of CPUs is varied. The SEQUENTIAL bar represents the unmodified benchmark running on a single core of an eight-core chip multiprocessor, while the two-CPU, four-CPU and eight-CPU bars represent the execution of full TLS-optimized executables running on two, four, and eight CPUs. Large improvements in transaction latency can be obtained by using two or four CPUs, although the additional benefits of using eight CPUs are small.



(a) PAYMENT (b) ORDER STATUS

Fig. 23. Performance of optimized benchmark while varying the number of CPUs.

Table III. Explanation of Graph Breakdown

Category	Explanation
Idle	Not enough threads were available to keep the CPUs busy.
Failed	CPU executed code that was later undone due to a violation (includes all time spent executing failed code.)
Latch stall	Stalled awaiting latch; latch is used when escaping speculation.
Cache miss	Stalled on a cache miss.
Busy	CPU was busy executing code.

Each bar is divided into subdivisions sthat show what each CPU is doing for each cycle of execution. The subdivisions are explained in Table III. In Figures 22 and 23 we have normalized the breakdown of time within each bar to the eight-CPU case so that the subdivisions of each bar can be directly compared. This means that the SEQUENTIAL breakdown shows one CPU executing and seven CPUs idling, the two-CPU breakdown shows two CPUs executing and six CPUs idling, etc.

The NEW ORDER, NEW ORDER 150, and DELIVERY OUTER bars show that very little time was spent on failed speculation—this means that our performance tuning was successful at eliminating performance-critical data dependences for those transactions. The DELIVERY transaction has some failed speculation due to a dependence in updating the ORDER LINE table, and the STOCK LEVEL transaction has failed speculation due to a dependence in the cursor used to scan the ORDER LINE table. 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 cache locality, since the execution is partitioned over more level 1 caches. We also see a much larger increase in the number of cache misses for the STOCK LEVEL transaction—a large amount of cache state can be invalidated when speculation fails, leading to increased cache misses. The negative effects of cache misses overwhelm any

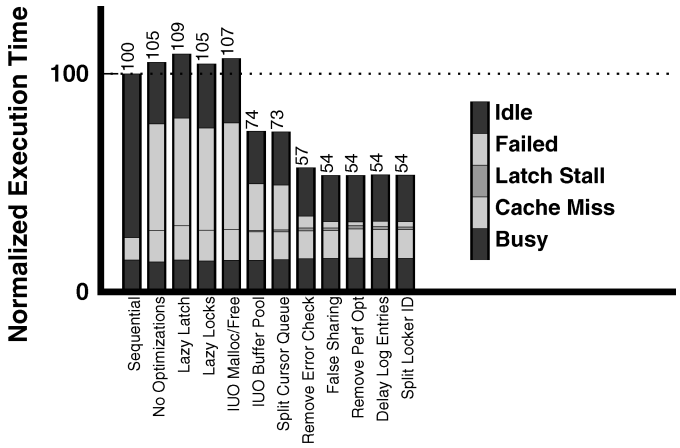
parallel overlap in the ORDER STATUS transaction, resulting in a slowdown as the number of CPUs increases.

The dominant component of the bars in NEW ORDER and DELIVERY is *idle* time, for three reasons. First, in the SEQUENTIAL, two-CPU and four-CPU case, we show the unused CPUs as idle to allow direct comparison with the other bars. Second, the loop that we parallelized in the transaction only covers 78% of the transaction's execution time for NEW ORDER, and 63% for DELIVERY: during the remaining time only one CPU is in use. Third, TPC-C specifies that both transactions will deal with orders that contain between 5 and 15 items, meaning 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 see the impact of reducing this idle time, we modified the invocation of the NEW ORDER transaction so that each order contains between 50 and 150 items (the NEW ORDER 150 transaction). We found that this modification decreases the amount of time spent idling, and does not significantly affect the trends in cache usage or failed speculation.

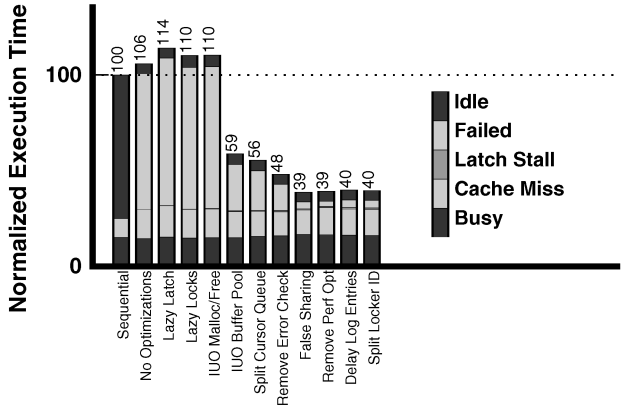
Figures 22 and 23 show a performance tradeoff: devoting more CPUs to executing a single transaction improves performance, but results in increased contention, a decrease in cache locality, and/or diminishing returns due to a lack of available parallelism, thus resulting in diminishing returns as more CPUs are added. One of the strengths of using TLS for intratransaction 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 that holds heavily contended locks. If many epochs are being violated, and thus the intratransaction 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 that are not already running in parallel, and periodically disable TLS for loops that are running in parallel. If the change improves performance then the scheduler can make it permanent.

5.5 Impact of Each Optimization

In Figure 24 we see the results of the optimization process for the benchmarks that sped up on a four CPU system. In this experiment, the breakdown of the bars is normalized to a four-CPU system, and so three-quarters of the SEQUENTIAL bars is *Idle*, since three of the four CPUs are idling during the entire execution. The NO OPTIMIZATIONS bars show what happens if we parallelize the transaction 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 between 5 and 8%.



(a) NEW ORDER

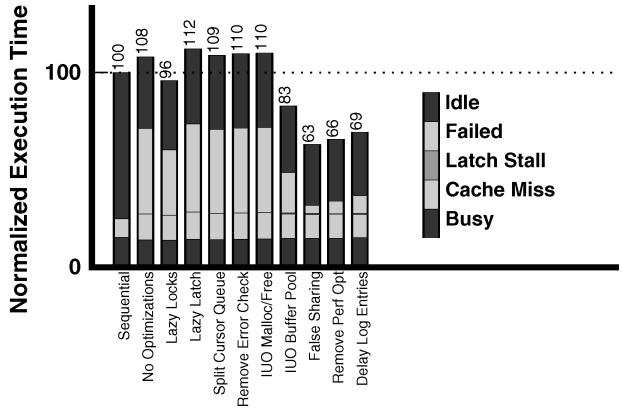


(b) NEW ORDER 150

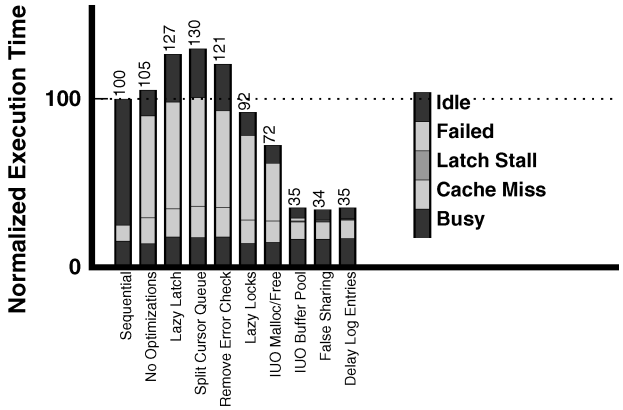
Fig. 24. Performance impact on the TPC-C transactions of adding each optimization one-by-one on a four-CPU machine.

Consider the NEW ORDER transaction in Figure 24(a). The major source of failed speculation 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 that is in the lock code. The first optimization also results in a slight slowdown, since the next bottleneck merely delays detection of failed speculation (as illustrated in Figure 10)—hence more execution has to be rewound.

Once we have eliminated latches as a bottleneck in NEW ORDER, 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 9 to `db_malloc` and the `pin_page` operation, parallelizing the free cursor pool, removing dependence causing error checks (Section 4.5), adding padding to avoid violations due



(a) DELIVERY

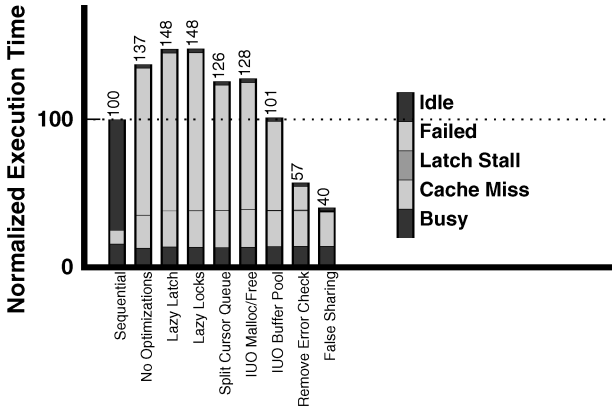


(b) DELIVERY OUTER

Fig. 25. Performance impact on the TPC-C transactions of adding each optimization one by one on a four-CPU machine.

to false sharing (Section 4.6), removing the "last page referenced" pointer from the B-tree search code (Section 4.3), delaying the generation of log entries until epochs are ready to commit (Section 4.2), and parallelizing the assignment of locker ids.

It is tempting to look at Figure 24(a) and conclude that the most important optimization was parallelizing the buffer pool, since adding this optimization caused the execution time to drop by 32%. 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 24 implies that the iterative optimization process that we used works well—as the database system programmer removes performance limiting dependences performance gradually improves (and exposes new dependences). Removing dependences decreases the time spent on failed execution, and improves performance.



(a) STOCK LEVEL

Fig. 26. Performance impact on the TPC-C transactions of adding each optimization one by one on a four-CPU machine.

Figure 24(b) shows the same experiment performed on the larger NEW ORDER 150 transaction. This transaction mirrors the NEW ORDER transaction, except the idling caused by load imbalance is no longer dominant with more epochs.

Figures 25(a) and 25(b) and 26 show the same experiment for the DELIVERY, DELIVERY OUTER, and STOCK LEVEL transactions. The order in which each bottleneck dependence becomes dominant varies from transaction to transaction. Also, not all of the bottlenecks found in NEW ORDER need to be removed to get the best performance out of these three transactions—for the STOCK LEVEL transaction performance actually degrades from 0.40 to 0.47 when the additional code to eliminate bottlenecks experienced by NEW ORDER is applied. STOCK LEVEL also shows that applying TLS can hurt performance: in the early rounds of optimization the transaction suffers dramatically from the decreased cache locality introduced by parallel execution, and there is insufficient parallel overlap to compensate for this effect. Overall, in DELIVERY, DELIVERY OUTER, and STOCK LEVEL the iterative process works quite well, resulting in significant performance improvements for each transaction.

We have shown an iterative optimization process in action. When should the iteration stop? Consider the FAILED segment of the bars in Figure 24. Note that eliminating a dependence avoids a violation, and hence only improves the performance of the FAILED portion of execution. The database system programmer chooses to stop when any potential gains in performance outweigh the perceived difficulty of eliminating the next bottleneck data dependence.

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 article, we have focused on one such opportunity enabled by TLS: exploiting *intratransaction* parallelism. Our experimental results demonstrate that we can speed up the *latency* (not just the throughput) of three of the five transactions in TPC-C by 44–66% by exploiting TLS on a chip multiprocessor with four CPU 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 intratransaction parallelism, we place almost no burden on the transaction programmer (they merely demarcate epoch boundaries). In the future this burden could easily be shifted to the transaction compiler. 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 will 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

Akkary, H. and Driscoll, M. 1998. A dynamic multithreading processor. In *Proceedings of MICRO-31*.

Arvind and Culler, D. 1986. Dataflow architectures. In *Annual Reviews in Computer Science*. Vol. 1. Palo Alto, CA. 225–253.

Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. 2000. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th ASPLOS*.

Bhowmik, A. and Franklin, M. 2002. A general compiler framework for speculative multithreading. In *Proceedings of the 14th SPAA*.

Colohan, C., Ailamaki, A., Steffan, J., and Mowry, T. 2006. Tolerating dependences between large speculative threads via sub-threads. In *Proceedings of the 33rd ISCA*.

Colohan, C. B. 2005. Applying thread-level speculation to database transactions. Ph.D. dissertation. Carnegie Mellon University, Pittsburgh, PA.

Colohan, C. B., Ailamaki, A., Steffan., J. G., and Mowry, T. C. 2007. CMP support for large and dependent speculative threads. *IEEE Trans. Paralled Distrib. Syst. 18*, 8, 1041–1054.

Eggers, S. and Jeremiassen, T. 1991. Eliminating false sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*. Vol. I. 377–381.

Franklin, M. and Sohi, G. 1996. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Trans. Comput. 45*, 5 (May), 552–571.

Garcia-Molina, H. and Salem, K. 1987. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY. 249–259.

Garzarán, M., Prvulovic, M., Llabería, J., Viñals, V., Rauchwerger, L., and Torrellas, J. 2003. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. In *Proceedings of the 9th HPCA*.

Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*. 15–26.

Gopal, S., Vijaykumar, T., Smith, J., and Sohi, G. 1998. Speculative versioning cache. In *Proceedings of the 4th HPCA*.

GRAY, J. 1993. *The Benchmark Handbook for Transaction Processing Systems*. Morgan-Kaufmann Publishers, San Francisco, CA.

GUPTA, M. AND NIM, R. 1998. Techniques for speculative run-time parallelization of loops. In *Proceedings of Supercomputing'98*.

HALSTEAD, JR., R. 1985. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Prog. Lang. Syst. 7*, 4, 501–538.

HAMMOND, L., CARLSTROM, B. D., WONG, V., HERTZBERG, B., CHEN, M., KOZYRAKIS, C., AND OLUKOTUN, K. 2004a. Programming with transactional coherence and consistency (TCC). In *Proceedings of the 11th ASPLOS*.

HAMMOND, L., HUBBERT, B., SIU, M., PRABHU, M., CHEN, M., AND OLUKOTUN, K. 2000. The Stanford Hydra CMP. *IEEE Micro. 20*, 2, 71–84.

HAMMOND, L., WONG, V., CHEN, M., CARLSTROM, B. D., DAVIS, J. D., HERTZBERG, B., PRABHU, M. K., WIJAYA, H., KOZYRAKIS, C., AND OLUKOTUN, K. 2004b. Transactional memory coherence and consistency. In *Proceedings of the 31st ISCA*.

HERLIHY, M. AND MOSS, J. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th ISCA*.

IBM CORPORATION. 2004. *IBM DB2 Universal Database Administration Guide: Performance*. IBM Corporation, Yorktown Heights, NY.

JEREMIASSEN, T. E. AND EGGERS, S. J. 1995. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *PPOPP'95: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 179–188.

JOHNSON, T., EIGENMANN, R., AND VIJAYKUMAR, T. 2004. Min-cut program decomposition for thread-level speculation. In *Proceedings of the ACM SIGPLAN'04 Conference on Programming Language Design and Implementation*.

KAUFMANN, H. AND SCHEK, H. 1996. Extending TP-monitors for intra-transaction parallelism. In *Proceedings of the 4th PDIS*.

KNIGHT, T. 1986. An architecture for mostly functional languages. In *Proceedings of the ACM Lisp and Functional Programming Conference*. 500–519.

KUNG, H. AND ROBINSON, J. 1981. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2, 213–226.

MAHLKE, S., CHEN, W., GYLLENHAAL, J., AND HWU, W. 1992. Compiler code transformations for superscalar-based high-performance systems. In *Proceedings of the International Conference on Supercomputing*.

MARCUELLO, P. AND GONZÁLEZ, A. 1999. Clustered speculative multithreaded processors. In *Proceedings of the ACM International Conference on Supercomputing*.

MARTÍNEZ, J. F. AND TORRELLAS, J. 2002. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA).

MCFARLING, S. 1993. Combining branch predictors. Tech. rep. TN-36. Digital Western Research Laboratory, Palo Alto, CA.

MCWHERTER, D., SCHROEDER, B., AILAMAKI, A., AND HARCHOL-BALTER, M. 2004. Priority mechanisms for OLTP and transactional Web applications. In *Proceedings of the IEEE International Conference on Data Engineering*.

MCWHERTER, D. T., SCHROEDER, B., AILAMAKI, A., AND HARCHOL-BALTER, M. 2005. Improving preemptive prioritization via statistical characterization of OLTP locking. In *Proceedings of the IEEE International Conference on Data Engineering*.

MILLER, J. AND LAU, H. 2001. *Microsoft SQL Server 2000 Resource Kit*. Chapter RDBMS: Performance Tuning Guide for Data Warehousing. Microsoft Press: Redmond, WA. 575–653.

MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst. 17*, 1, 94–162.

MORRISETT, G. AND HERLIHY, M. 1993. Optimistic parallelization. Tech. rep. CMU-CS-93-171. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

OLSON, M., BOSTIC, K., AND SELTZER, M. 1999. Berkeley DB. In *Proceedings of the Summer Usenix Technical Conference*.

OLUKOTUN, K., HAMMOND, L., AND WILLEY, M. 1999. Improving the performance of speculatively parallel applications on the hydra CMP. In *Proceedings of the 13th Annual ACM International Conference on Supercomputing*.

OOI, C. L., KIM, S. W., PARK, I., EIGENMANN, R., FALSAFI, B., AND VIJAYKUMAR, T. N. 2001. Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *Proceedings of the International Conference on Supercomputing*.

OPLINGER, J., HEINE, D., AND LAM, M. 1999. In search of speculative thread-level parallelism. In *Proceedings of PACT '99*.

PRABHU, M. AND OLUKOTUN, K. 2003. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the ACM SIGPLAN 2003 Symposium on Principles & Practice of Parallel Programming*.

PRVULOVIC, M., GARZARÁN, M. J., RAUCHWERGER, L., AND TORRELLAS, J. 2001. Removing architectural bottlenecks to the scalability of speculative parallelization. In *Proceedings of the 28th ISCA*.

RAJWAR, R. AND GOODMAN, J. 2001. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*.

RAUCHWERGER, L. AND PADUA, D. 1999. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans. Parallel Distrib. Syst. 10*, 2, 160–172.

ROTENBERG, E., JACOBSON, Q., SAZEIDES, Y., AND SMITH, J. 1997. Trace processors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*.

RUNDBERG, P. AND STENSTROM, P. 2000. Low-cost thread-level data dependence speculation on multiprocessors. In *Proceedings of the Fourth Workshop on Multithreaded Execution, Architecture and Compilation*.

RYS, M., NORRIE, M., AND SCHEK, H. 1996. Intra-transaction parallelism in the mapping of an object model to a relational multi-processor system. In *Proceedings of the 22nd VLDB*.

SHASHA, D., LLIRBAT, F., SIMON, E., AND VALDURIEZ, P. 1995. Transaction chopping: Algorithms and performance studies. *ACM Trans. Database Syst. 20*, 3, 325–363.

SHINNAR, A., TARDITI, D., PLESKO, M., AND STEENSGAARD, B. 2004. Integrating support for undo with exception handling. Tech. rep. MSR-TR-2004-140. Microsoft Research. Redmond, WA.

SILBERSCHATZ, A., GALVIN, P., AND GAGNE, G. 2002. *Operating System Concepts*. John Wiley & Sons, New York, NY.

SOHI, G., BREACH, S., AND VIJAYKUMAR, T. 1995. Multiscalar processors. In *Proceedings of the 22nd ISCA*.

STEFFAN, J., COLOHAN, C., AND MOWRY, T. 1997. Architectural support for thread-level data speculation. Tech. rep. CMU-CS-97-188. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

STEFFAN, J., COLOHAN, C., ZHAI, A., AND MOWRY, T. 2000. A scalable approach to thread-level speculation. In *Proceedings of ISCA 27*.

STEFFAN, J., COLOHAN, C., ZHAI, A., AND MOWRY, T. 2002. Improving value communication for thread-level speculation. In *Proceedings of the 8th HPCA*.

STEFFAN, J. AND MOWRY, T. 1998. The potential for using thread-level data speculation to facilitate automatic parallellization. In *Proceedings of the 4th HPCA*.

STEFFAN, J. G., COLOHAN, C. B., ZHAI, A., AND MOWRY, T. C. 2005. The stampede approach to thread-level speculation. *ACM Trans. Comput. Syst. 23*, 3 (Aug.), 253–300.

TORRELLAS, J., LAM, M., AND HENNESSY, J. 1990. Shared data placement optimizations to reduce multiprocessor cache miss rates. In *Proceedings of the 1990 International Conference on Parallel Processing*. Vol. II. 266–270.

TRANSACTION PROCESSING PERFORMANCE COUNCIL. 2005. TPC benchmark C standard specification revision 5.4. Go online to `http://www.tpc.org`.

TREMBLAY, M. 1999. MAJC: Microprocessor architecture for Java computing. In *Proceedings of HotChips '99*.

VIJAYKUMAR, T. 1998. Compiling for the multiscalar architecture. Ph.D. dissertation. University of Wisconsin-Madison, Madison, WI.

YEAGER, K. 1996. The MIPS R10000 superscalar microprocessor. *IEEE Micro 16*, 2, 28–40.
ZHAI, A., COLOHAN, C., STEFFAN, J., AND MOWRY, T. 2002. Compiler optimization of scalar value communication between speculative threads. In *Proceedings of the 10th ASPLOS*.
ZHAI, A., COLOHAN, C., STEFFAN, J., AND MOWRY, T. 2004. Compiler optimization of memory-resident value communication between speculative threads. In *Proceedings of the International Symposium on Code Generation and Optimization*.
ZHANG, Y., RAUCHWERGER, L., AND TORRELLAS, J. 1999. Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors. In *Proceedings of the 5th HPCA*. 135–141.
ZUZARTE, C. 2005. Personal communication.

Received July 2006; revised August 2007; accepted November 2007