Applying Thread-Level Speculation to Database Transactions

Christopher B. Colohan

CMU-CS-05-188

November 2005

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Todd C. Mowry, Chair
Anastassia Ailamaki, Chair
Seth C. Goldstein
David O'Hallaron
Kunle Olukotun, Stanford University

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Aeronautics and Space Administration (NASA) under grant nos. NAG2-1230 and NAG2-6054, and by generous donations from the Intel Corporation, IBM Corporation, and the NSF.

Keywords: TLS, Threads, Speculation, Architecture, Database Systems, Transactions, Intra-Transaction Parallelism, Chip Multiprocessors, Cache design

For Lea.

Abstract

Thread-level speculation (TLS) is a promising method of extracting parallelism from both integer and scientific workloads. In this thesis we apply TLS to exploit *intra-transaction* parallelism in database workloads. Exploiting intra-transaction parallelism without using TLS in existing database systems is difficult, for two reasons: first, significant changes are required to avoid races or conflicts within the DBMS, and second, adding threads to transactions requires a high level of sophistication from transaction programmers. In this thesis we show how dividing a transaction into speculative threads (or *epochs*) 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. We also show that previous hardware support for TLS is insufficient for the resulting large speculative threads and the complexity of the dependences between them. In this thesis we extend previous TLS hardware support in three ways to facilitate large speculative threads: (i) we propose a method for buffering speculative state in the L2 cache, instead of solely using an extended store buffer, L1 data cache, or specialized table to track speculative changes; (ii) we tolerate cross-thread data dependences through the use of *sub-epochs*, significantly reducing the cost of mis-speculation; and (iii) with programmer assistance we *escape speculation* for database operations which can be performed non-speculatively. With this support we can effectively exploit intra-transaction parallelism in a database and dramatically improve transaction performance: on a simulated 4-processor chip-multiprocessor, we improve the response time by 46–66% for three of the five TPC-C transactions.

Acknowledgments

My first advisor was Todd Mowry. He brought me to Pittsburgh, taught me how to do research, stuck with me through many challenging years, and provided the inspiration to finish. Once I finally figured out that database systems are the perfect application for TLS, Natassa Ailamaki was there to teach me everything I needed to know about databases, provide much needed encouragement, and provide loads of guidance and advice. Thank you very much to both of you, without your help this thesis would not exist.

Seth, Dave, and Kunle served on my thesis committee. Thank you for your probing questions and extensive advice which made this thesis much more solid than it would otherwise be.

I certainly would not have stuck with graduate school to completion if it wasn't for the CS department at CMU. The wonderful community formed by the faculty and students there provide much needed support through the entire process. I would especially like to thank **Catherine Copetas** and **Sharon Burks** for providing friendship, abuse, and help throughout my grad school years.[1]

I would like to say "I did it all myself! This thesis is mine, mine, all mine!" Instead, I can say something even better: this thesis grew out of a collaboration. The ideas used in TLS grew out of the STAMPede project, which was created by Greg Steffan, Antonia Zhai, Todd Mowry and myself. We designed hardware, built simulators, programmed compilers, and argued over ideas. We drew upon the input of the rest our research group, including Amit, Pedro, Shimin and Mengzhi. We learned from the efforts of our gifted colleagues at other universities. This thesis is just a small facet of this entire project, and could not exist in isolation. Thank you all.

I would also like to thank my friends, who helped keep me sane throughout school. You have all touched me in so many ways: friends, climbing partners, dancers, roommates, soul mates, fellow musicians, chefs, accomplices... I fear the prospect of listing you, for I know I will inevitably leave someone out. Here goes: Ted, Addie, Leaf, Chris, Carrie,

[1]Why the large font? Catherine said that when I mentioned Sharon and her I should put their names in a larger font to make it easier for everyone to find them. The things I put up with...

Dave, Hope, Lily, Angela, Joe, Jason, Greg, Nancy, Donna, Dan, Yan, Francisco, Will, Benoit, Elissa, Paul, Rob, Caitlin, Dean, Aaron, Jenn, Rob, Peter, Andy, Charlie, Dominic, Anya, Rose, Derek, Ricardo, Rowan, Linda, Mark, Emily, Carrie, and the rest of the Los Bailadores dance team. Thank you all.

Thank you to my family, and especially my parents, for providing years of support and encouragement: I can not honestly answer the question of "when are you going to finish?" The answer is: **now**.[2]

Last, but certainly not least: thank you to my fiancée, Lea. You are my love.

[2]Yes, the word "now" was the last word I typed in the creation of this entire document.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We are in the midst of a revolution in microprocessor design: all of the major computer manufacturers are producing computer systems that feature chip multiprocessors (CMPs) and simultaneous multithreading (SMT). Examples include Intel's "Smithfield" (dual-core Pentium IV's with 2-way SMT), IBM's Power 5 (combinable, dual-core, 2-way SMT processors), AMD's Opteron (dual-core), and Sun Microsystems's Niagara (an 8-processor CMP). How can database systems exploit this increasing abundance of hardware-supported threads? Currently, for OLTP workloads, threads are primarily used to increase transaction *throughput*; ideally, we could also use these parallel resources to decrease transaction *latency*. Although most commercial database systems do exploit *intra-query parallelism* within a transaction, this form of parallelism is only useful for long running queries, while OLTP workloads tend to issue multiple short queries. To the best of our knowledge, commercial database systems do not exploit *intra-transaction parallelism* [24, 37, 69], and for good reason.

Parallelizing a transaction is difficult. First, the DBMS must be modified to support multiple threads per transaction. Latches[1] (a.k.a. mutexes) must be added to data structures which are shared between threads in the transaction. These latches add complexity and hinder performance. Second, the transaction must be divided into parallel threads. Consider the NEW ORDER transaction, which is the prevalent transaction in TPC-C [15] (Figure 1.1). 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 interfere with each other, and add inter-thread locks to avoid problems; e.g., the thread should use inter-thread locks to ensure that only one thread updates the quantity of an item in the `stock` table at a time. Finally, the transaction

[1]A latch provides mutual exclusion between two threads. Latch is a database systems term, a latch is known as a *mutex* in operating systems parlance.

```
begin_transaction {
    Read customer info [customer, warehouse]
    Read & increment order # [district]
    Create new order [orders, neworder]
    for(each item in order){                          ⎫
        Get item info [item]                           ⎪
        if(invalid item)                               ⎪
            abort_transaction                          ⎪
        Read item quantity from stock [stock]          ⎬ 78% of transaction
        Decrement item quantity                        ⎪ execution time
        Record new item quantity [stock]               ⎪
        Compute price                                  ⎪
        Record order info [order_line]                 ⎪
    }                                                  ⎭
} end_transaction
```

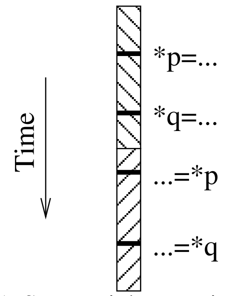
Figure 1.1: The NEW ORDER transaction. In brackets are the database tables touched by each operation.

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.

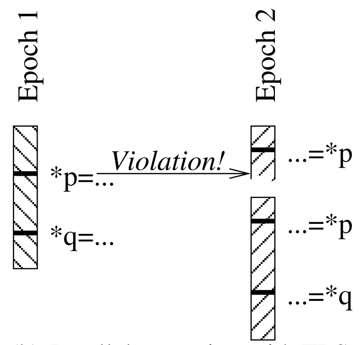
Significant effort is required to add appropriate latches to the DBMS and transaction code. Once synchronization is added, significant tuning must be performed to avoid over-synchronization. Can the process of extracting parallelism from transactions be automated?

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 problem with this conservative programming model is that the initial assumption is that everything is run in parallel, and then the programmer must add in latches and locks to preserve correctness. If the programmer does not has have a deep understanding of the entire program then they will either unknowingly introduce bugs by omitting needed locks, or reduce performance by adding unnecessary locks.

To avoid burdening the programmer, can we can automatically detect data dependences as they occur at run time? Out-of-order CPUs can detect the fine-grained data dependences



(a) Sequential execution.



(b) Parallel execution with TLS.

Figure 1.2: How TLS ensures that all reads and writes occur in the original sequential order.

between individual instructions at run time to extract *instruction level parallelism* (ILP). We would like to automate dependence checking at the granularity of *thread-level parallelism* (TLP) as well.

1.1 Thread-Level Speculation

Thread-Level Speculation (TLS) [18, 60, 63] provides a middle ground between programmer intensive explicit parallelization and ILP. With TLS, the programmer specifies where to break a transaction into threads, or *epochs*,[2][3] 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. In essence, dividing a transaction into epochs improves performance without affecting correctness.

Under TLS, sequential code (Figure 1.2(a)) is divided into *epochs*, which are executed in parallel by the system (Figure 1.2(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`).

[2]We refer to the parallel threads in TLS as *epochs* to differentiate them from the explicit threads which 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.

[3]There are many new terms used in this thesis. Please note that there is a *Glossary* located at the back of the thesis for your convenience.

The system observes whether epoch 1 ever writes to a memory location which has already been read by epoch 2—if so, then epoch 2 has *violated* sequential semantics, and is rewound and re-executed with the correct value. For example, in Figure 1.2(b) we see that epoch 2 read `p` before epoch 1 wrote to `p`, so we restart epoch 2. On the second execution epoch 2 reads the new value. Note that the read of `q` does not cause a violation, since it executes *after* the write to `q`, and thus reads the correct value. By observing all loads and stores, and restarting an epoch whenever it consumes an incorrect value, the TLS mechanism ensures that the parallel execution is identical to the original sequential execution.

To support TLS-style execution, the system must have two capabilities. First, the system must be able to *detect* mis-speculation caused by data dependences between epochs. Once detected, the dependence triggers a violation which rewinds the epoch. Second, the system must *buffer* any changes to the system state performed by an epoch, so if the epoch is violated then the changes can be rewound.

Prior to the work in this thesis, if TLS were applied to the main loop of a transaction then there would be *no* performance gain. This is because:

1. The epochs in the transaction frequently invoke the database management system (DBMS). The DBMS performs many operations on shared data structures, and this means that each epoch has many data dependences on prior epochs.

2. TLS restarts an epoch when a violation is detected. With frequent violations only the oldest epoch is never restarted. This results in the oldest epoch making forward progress while the other epochs frequently restart—which results in an execution very similar to sequential execution.

3. The epochs in the transaction are large, which means that the TLS system has to buffer a large amount of speculative state. Prior TLS designs assumed that the speculative state would fit into small buffers, and do not perform well when speculative buffers overflow.[4]

This thesis describes how small changes can be made to the DBMS to avoid the most problematic data dependences, how to modify the TLS hardware to be more tolerant of frequent data dependences, and how to modify the TLS hardware to buffer large amounts of speculative state. The result is a significant latency improvement for database transactions—on a simulated 4-processor chip-multiprocessor, the response time is improved by 46–66% for three of the five TPC-C transactions. Fundamentally, this thesis answers the question

[4]Prvulovic *et. al.* have developed hardware techniques [47] to buffer the large epochs found in scientific workloads, but their design does not perform well when epochs frequently restart.

"how can we parallelize the central loop of a transaction?" More formally, the thesis statement is:

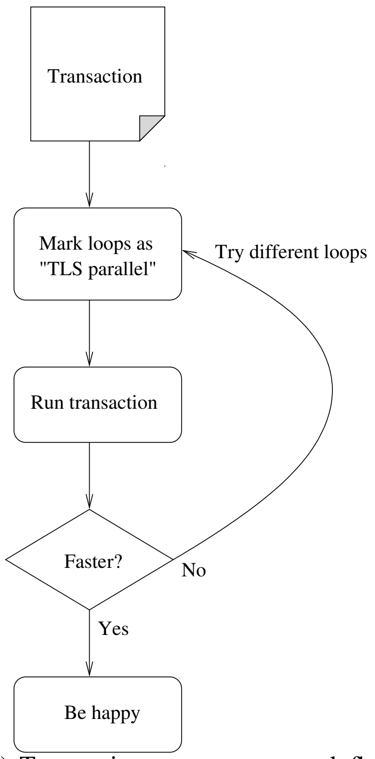
> *Thread-level speculation can be used to parallelize database transactions. This effectively reduces transaction latency without changing the transactions software, with minor changes to the DBMS software, and with practical hardware support.*

1.2 Small Software Changes for Large Performance Gains

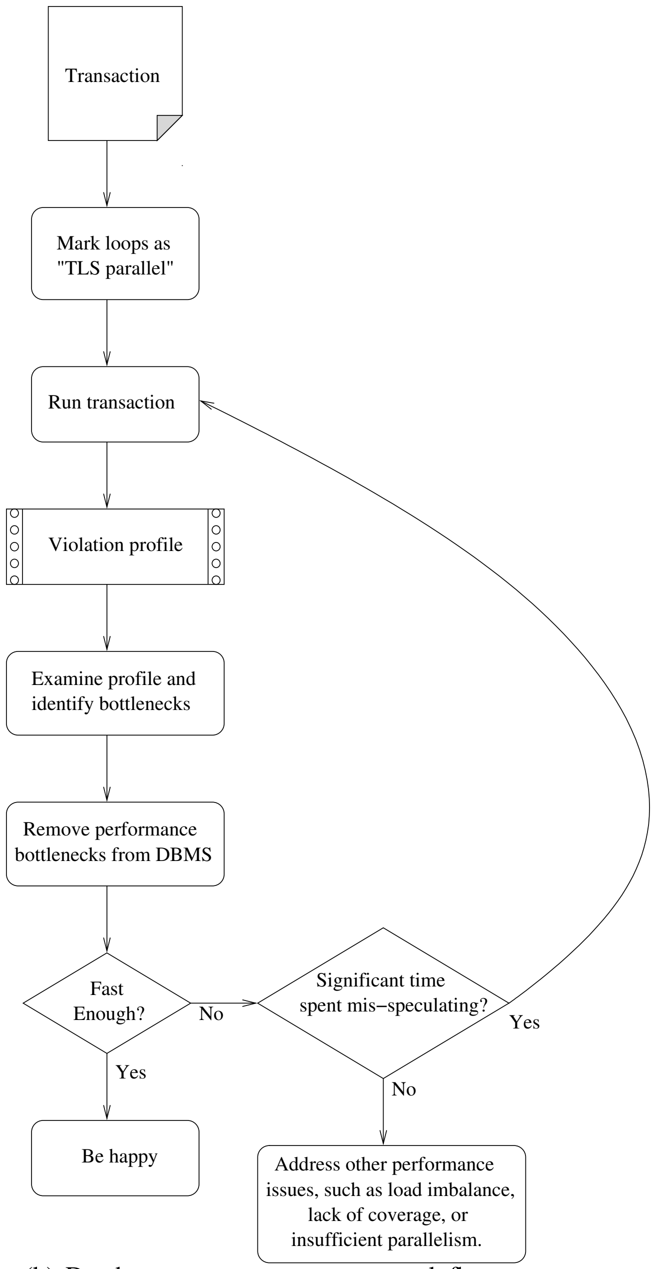
To achieve these impressive performance benefits we need to modify both the transaction code and the code in the DBMS. For TLS to be widely adopted, these software changes must be reasonable.

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 1.3(a)). TLS is a black box which can add parallelism to transaction without changing the original sequential execution semantics, which means that TLS will not introduce any new bugs (although the change in performance may trigger race conditions which already existed in the original code). The interface to TLS used by the transaction programmer can be simple: they provide a performance hint, 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 which modify large portions of the database system. To apply TLS, the database system programmer engages in an iterative process (Figure 1.3(b)). They start with a sequential transaction, mark some loops for TLS execution, and then examine the profile information provided by the hardware. This profile information says which loads and stores caused dependence violations, and hence are reducing performance. The database system programmer then optimizes *just this performance critical code*, and repeats the process. This thesis shows that this tuning process requires modifying only a small fraction of the DBMS code (we only modified 1200 out of 180,000 lines of code in BerkeleyDB), and yet eliminates the majority of the data dependences which cause violations. This means that the gains of TLS can be had with a small effort on the part of the database vendor.



(a) Transaction programmer work flow.


Transaction
Mark loops as "TLS parallel"
Run transaction
Violation profile
Examine profile and identify bottlenecks
Remove performance bottlenecks from DBMS
Fast Enough?
No
Yes
Be happy
Significant time spent mis−speculating?
Yes
No
Address other performance issues, such as load imbalance, lack of coverage, or insufficient parallelism.

(b) Database system programmer work flow.

Figure 1.3: How the transaction programmer and database system programmer use TLS.

1.3 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 [35]. If the system is lock-bound, then to improve performance we need to decrease the time spent waiting for locks. TLS can be used to improve the latency of the transaction which holds the contended locks, and hence it will release the contended locks more quickly. This latency improvement causes an improvement in transaction throughput [36].

1.4 Thesis Overview

This thesis explains the application of TLS to databases in a top-down fashion. In Chapter 2 we explain the fundamentals of TLS, as viewed by the user of TLS. At the end of Chapter 2 the reader should understand how to divide an application up into epochs, how epochs interact with each other as they execute, the performance impact of data dependences, and what mechanisms TLS can provide to avoid performance limiting data dependences (value forwarding, aggressive update propagation, sub-epochs, isolated undoable operations, delaying until arrival of homefree token).

Once we have thoroughly explained how TLS works, the thesis proceeds to apply TLS to database transactions. Chapter 3 walks the reader through the process of applying TLS to transactions from the TPC-C benchmark, and shows that even if the transaction appears to be fully parallel, the database management system (DBMS) functions invoked by the transaction are not. We fix this one step at a time, through a iterative performance tuning process. This tuning process involves changes to all of the major DBMS subsystems, including locking, B-tree management, logging, performance monitoring, buffer pool management, and memory allocation. Chapter 3 shows how TLS can be used to improve the latency of transactions.

Chapter 3 presupposes that hardware which can handle the execution of large epochs with many dependences between them exists. In Chapter 4 we show how to design TLS hardware which can deal with large epochs with many dependences. The hardware design in Chapter 4 has three unique aspects: first, speculative state is stored in multiple levels of the cache hierarchy. Second, the speculation mechanism can be temporarily disabled to allow software to manage speculation when desired. Third, dependences between epochs are *tolerated*, and the hardware includes mechanisms which avoid mis-speculation and reduce the penalty of mis-speculation when it occurs.

1.5 Related Work

The allure of parallelizing programs has attracted many software and hardware researchers. Parallel programs should be able to perform much better than sequential programs, since they can harness the power of many CPUs at once. The work which inspired and influenced this thesis can be divided into three categories: research in parallel programming models, research in intra-transaction parallelism, and research in TLS hardware support.

1.5.1 Parallel Programming Models

Decades of work have been invested in parallelizing compilers and parallel programming languages. TLS promises the holy grail of parallel performance—to allow any program, no matter what language it is written in, to be automatically transformed into a full parallel program with no programmer effort. As a result, most TLS research has assumed that the *compiler* would be parallelizing programs, and not a human [5, 25, 44, 65]. This compiler work grew out of the pioneering work of Knight and Halstead, who developed functional languages with support for TLS-style execution [17, 28]. In this thesis we wanted to parallelize database transactions. Database transactions and the DBMSs they utilize are not written in functional languages, and it is not practical to re-implement 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. In this thesis 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 thesis 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 [18, 46]. Prabhu and Olukotun's work gave us hope that applying TLS to the much larger epochs from the outer loops of database transactions would be worthwhile as well.

Hammond *et. al.* pushes the idea of programming with epochs to its logical extreme, making the program consist of nothing but epochs, resulting in a simpler architecture [20], but requires the programmer to always use epochs [19]. We believe that it is not practical to apply this approach to existing database systems. As a result the work in this thesis 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. The price paid for this programmer simplicity and flexibility is a minor increase in hardware complexity.

1.5.2 Intra-transaction Parallelism

Traditionally, high-performance database systems have targeted inter-transaction parallelism, or intra-operation parallelism, while this thesis introduces new techniques for exploiting intra-transaction parallelism. Previous work on intra-transaction 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 sagas to abort the transaction 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 [27, 52], 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.*, who developed a theoretical basis for automatically breaking a saga into transactions. Shasha 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 [53]. 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.

1.5.3 TLS Hardware Support

The basic idea behind TLS is inspired by Kung and Robinson's optimistic concurrency control (OCC) work [29]. In Kung and Robinson's paper they propose 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 thesis 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 [23], 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 [39]. The hardware design in this thesis 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 [9, 55]. 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 [9]. Later this design was refined to use the caches to detect and buffer memory dependences, in the form of the speculative versioning cache [14]. The success of the Multiscalar project inspired numerous other TLS research projects, including the IACOMA project [47], the Hydra project [18], and our Stampede project [58, 59, 60, 61, 67, 68]. This research also inspired work on a few software-only TLS designs [16, 49, 51] and hardware-only TLS designs [1, 32, 50]. An interesting comparison of many of these schemes was done by Garzarán *et. al.* [11].

One of the ways which we avoid dependences in this thesis 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 [23, 29]: (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 non-speculative transactions with *no* changes to the non-speculative transactions.

This thesis adds an important capability to prior hardware designs: we use sub-epochs to tolerate data dependences between speculative epochs. Sub-epochs are a form of checkpointing, and in this thesis sub-epochs are used to reduce the penalty due to failed speculation. Using checkpoints in epochs was previously proposed by Olukotun *et. al.* [43]—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 sub-epochs (checkpoints). Prior work has also used checkpointing to simulate an enlarged reorder buffer by storing multiple checkpoints in the load/store queue [2, 7], and a single checkpoint in the cache [33]. Martinez's checkpointing scheme [33] effectively enlarges the reorder buffer and is also integrated with TLS, and is thus it is unsurprising that his hardware design is close to the design in this thesis. The sub-epoch design in this thesis could be used to provide a superset of the features in Martinez's work: sub-epochs could provide multiple checkpoints with a large amount of state in the L2 cache. Martinez's scheme, being simpler and involving only the L1 cache, would likely be faster. Tuck and Tullsen showed how thread contexts in a SMT processor could be used to checkpoint the system and recover from failed value prediction, expanding the effective instruction window size [64]—the techniques used to create sub-epochs in this thesis could also be used to create checkpoints at high-confidence prediction points using the techniques from Tuck's thesis. Instead of using sub-epochs to tolerate data dependences other studies have explored predicting data dependences and turning them into synchronization [40, 61], or have used the compiler to mark likely dependent loads and tried to predict the consumed values at run time [61]. It is not practical to use these predictors instead of sub-epochs for the large epochs examined in this thesis: the epochs in this thesis have many dependences between them, which means that the predictors would have to be extremely accurate to avoid mis-predictions which restart the entire epoch. On the other hand, using sub-epochs significantly reduces the cost of mis-speculation, which means that applying these prediction techniques may improve the results in this thesis even further.

To implement both shared cache TLS support and sub-epochs we store multiple versions of values in the cache. Multiple versions are also supported in Speculative Versioning Cache (SVC) and the IACOMA [14, 47] approach. In this thesis we show how to store speculative versions of lines in multiple levels of the cache hierarchy, allowing us to take advantage of the larger sizes and associativities of the caches further from the CPUs. By limiting the visibility of replicas to just the CPUs sharing an L2 cache, we avoid the need for the version ordering list used in SVC. In this thesis we detail a design for a speculative victim cache, which we find is sufficient to capture cache set overflows for our database applications. The IACOMA speculative buffer overflow technique [47] or techniques developed for supporting large transactions in transactional memory [3, 48] can be used if victim cache overflow becomes an issue.

1.6 Contributions

This thesis has three primary contributions. This thesis:

1. Provides a programming methodology which lets a programmer start with *correct* sequential code, and incrementally modify the code to create *correct* parallel code with a minimal amount of programmer effort.

2. Demonstrates the applicability of this methodology by using it to parallelize the central loop of database transactions.

3. Increases the scope and applicability of this methodology by extending TLS hardware designs to support incremental improvements of large, partially-dependent epochs.

Chapter 2

A New TLS Programming Model for Large, Dependent Epochs

To use TLS, the first step is to divide the transaction into epochs. The programmer or compiler must choose epochs which will result in a performance improvement. In this chapter we start with a brief review of what epochs are, how they interact with each other. We then introduce new 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 2.1, 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.

But what about the `select` and `update` functions? If we look at the definitions of those functions (also in Figure 2.1), we see that they increment the variables `num_selects` and `num_updates`. The increments will cause *data dependences* between our epochs, shown in Figure 2.2(a). In addition, the `select` and `update` functions call `btree_lookup`, and we do not know what dependences may exist in the B-tree code.

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 2.2(a) we show sev-

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

Each iteration is an epoch

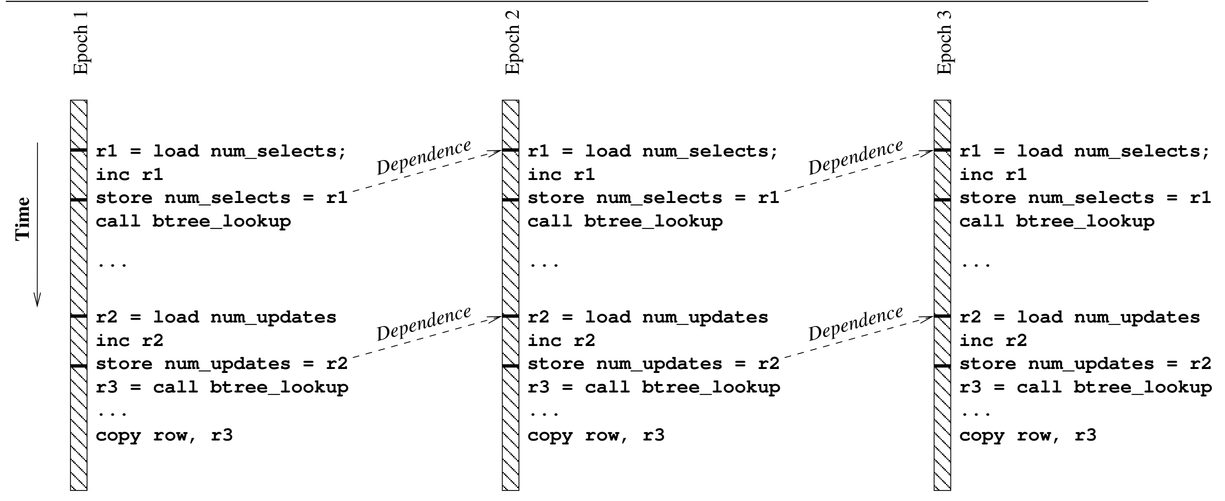
```
Row
StockTable::select(ItemId item)
{
  num_selects++;
  return *btree_lookup(item);
}

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

Figure 2.1: Simplified main loop from the NEW ORDER transaction.

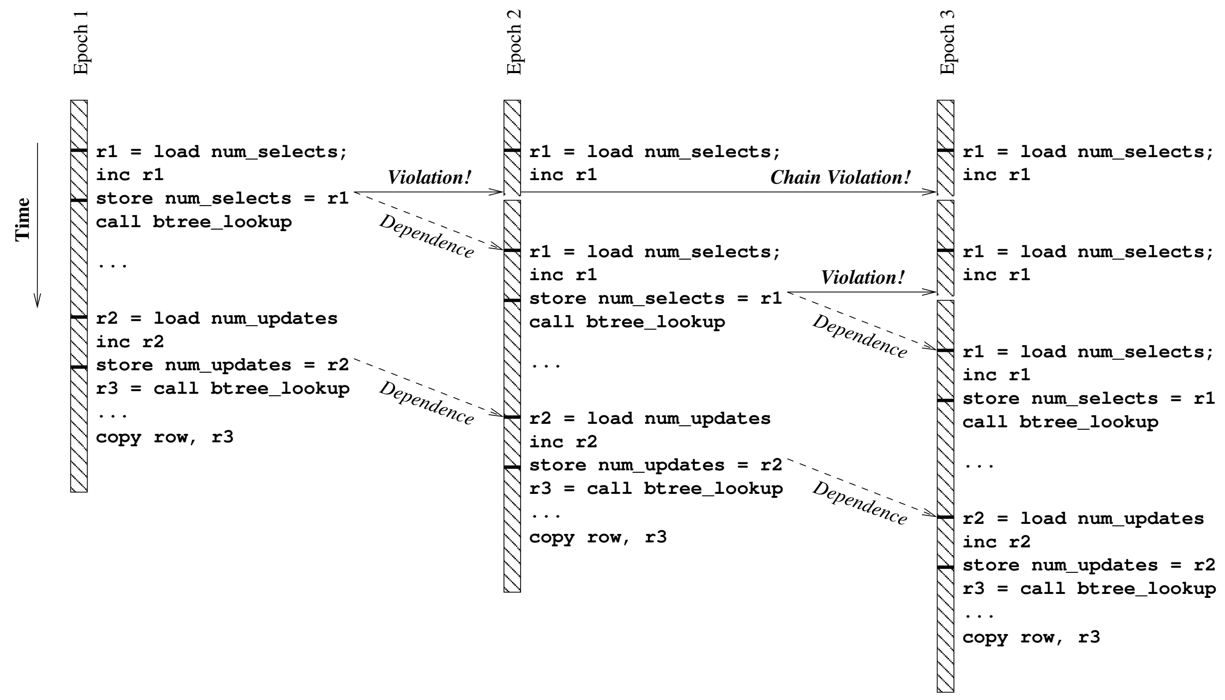
eral *backwards dependences*, where the load executes before the last store in the previous epoch executes. When TLS detects a backwards dependence it is known as a *dependence violation*, and it restarts the epoch (Figure 2.2(b)). [1] In addition to restarting the epoch, all later epochs are restarted as well through a *chain violation* since they may have consumed incorrect speculative values from the violated epoch. When the violated epoch re-executes, the backwards dependence that triggered the violation is turned into a forwards dependence, and hence the correct result is computed.

Since violations cause work to be discarded and re-executed, violations limit perfor-

[1]In this discussion we assume that only backwards dependences can cause violations. This is because the TLS hardware described in Chapter 4 makes an epoch's stores (updates) available to later epochs as soon as possible. This is known as *aggressive update propagation.* Using 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).



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



(b) Backwards dependences cause mis-speculation. Violations recover from mis-speculation by restarting the epoch.

Figure 2.2: TLS execution of the example loop.

mance. To mitigate this performance hit, prior work has done the following:

Limit epoch sizes. Restarting large epochs throws away a large amount of work, which is not efficient. 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. [65]

Choose epochs to avoid *backwards* dependences. Sometimes dependences can not be avoided completely—in these cases it makes sense to either choose epochs to try and avoid backwards dependences, or to apply compiler scheduling techniques to the epochs in an attempt to turn backwards dependences into forwards dependences. [65, 68]

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. [68]

Use hardware managed synchronization. If hardware detects a load-store pair which causes frequent violations, insert synchronization which ensures that the load does not issue until the store retires. [61]

Use value prediction. Detect which loads frequently cause violations, and use a value predictor to predict the correct value consumed by the load. [45]

To parallelize database transactions, we started with a simple division into epochs—we made each loop iteration in the transaction into an epoch. We chose a simple division to minimize the amount of work that is performed by the transaction programmer. The resulting epochs had the following properties:

- The epochs were large. With epochs much larger than previous work had studied, we found that violations discarded huge amounts of execution. Previous work has studied epochs with various size ranges, including 3.9–957.8 dynamic instructions [65], 140–7735 dynamic instructions [46], 30.8–2,252.7 dynamic instructions [61], and up to 3,900–103,300 dynamic instructions [11]. The epochs studied in this thesis are quite large, with 7,574–489,877 dynamic instructions. Large epochs result in lots of wasted work when a violation occurs.

- There were many violation causing dependences between epochs.

- The epochs were not amenable to compiler based synchronization techniques. The majority of dependences were caused by 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 prediction and synchronization techniques. Each epoch in our database transactions had tens to hundreds of data dependences which would have to be handled. A mis-prediction triggers a violation, which rewinds the entire epoch. A very accurate predictor would be required to be able to reliably perform tens to hundreds of predictions in a row without making a single error.

Clearly something had to be done. We did not want to change the selection of epochs, since keeping epoch selection simple makes using TLS much more attractive for transaction programmers. We instead adopted a two pronged approach:

1. Modify the DBMS to avoid or remove data dependences which frequently trigger violations. In this chapter we examine how removing or avoiding dependences affects performance; Chapter 3 applies these techniques to the DBMS to remove dependences.

2. Modify the hardware so that violations do *not* cause the entire epoch to be rewound. This is done by dividing the epoch into *sub-epochs*. We explain how sub-epochs work in this chapter, in Chapter 4 we explain how they are implemented in hardware.

2.2 Sub-epochs

Data dependences between epochs cause failed speculation, which limits performance. If the transaction was executing on a dataflow architecture [4] then modifying or re-arranging 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 2.3 we see that if a programmer eliminates a dependence early in the epoch's execution then it may *hurt* performance by exposing a dependence which occurs later in the epoch's execution.

Performance is hurt because on a violation the TLS mechanism rewinds both the misspeculated execution which depends on the errant load *and* all of the correct execution which precedes the errant load. When a dependence occurs late in the epoch's execution

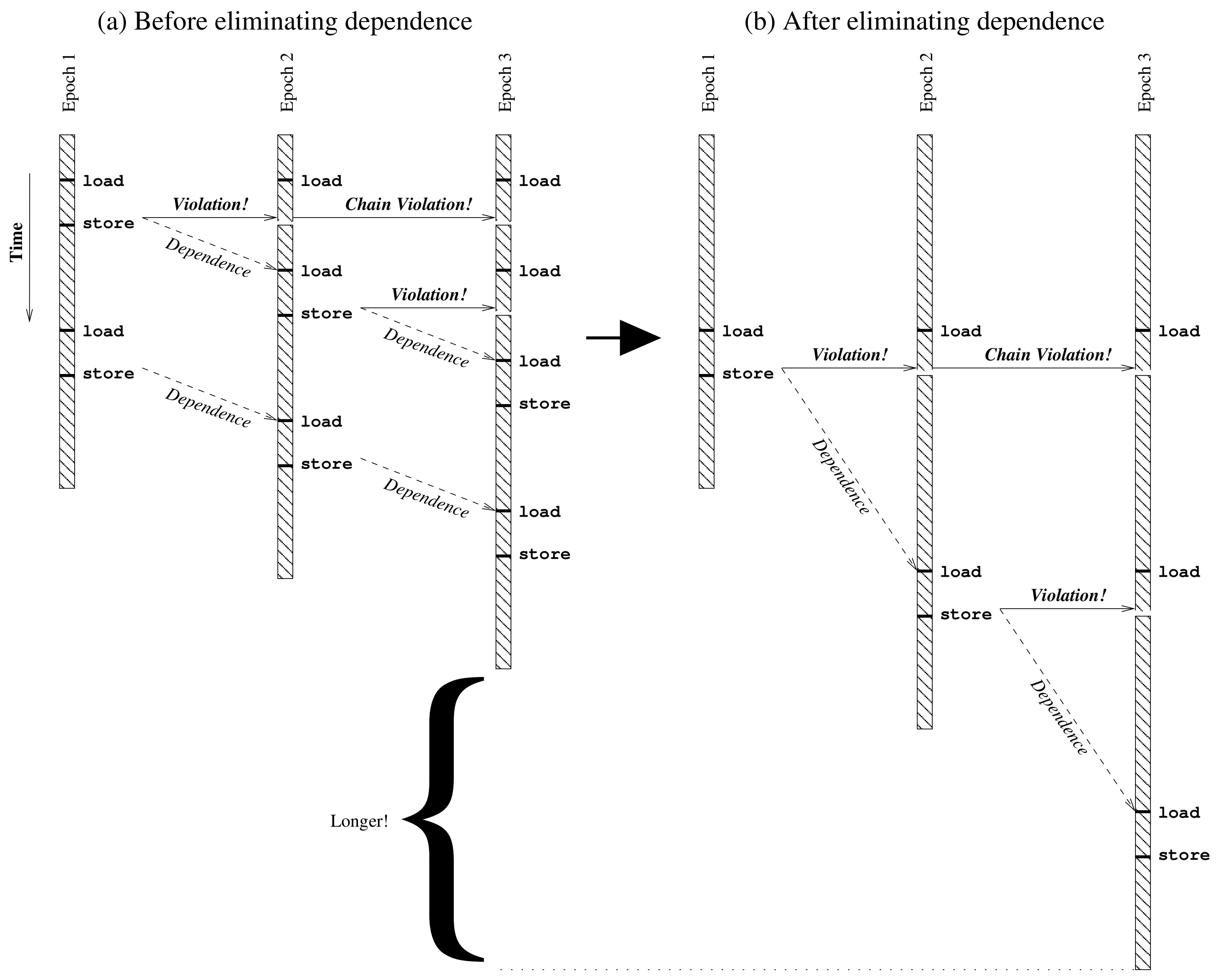


Figure 2.3: Eliminating the first dependence in an epoch can *hurt* performance.

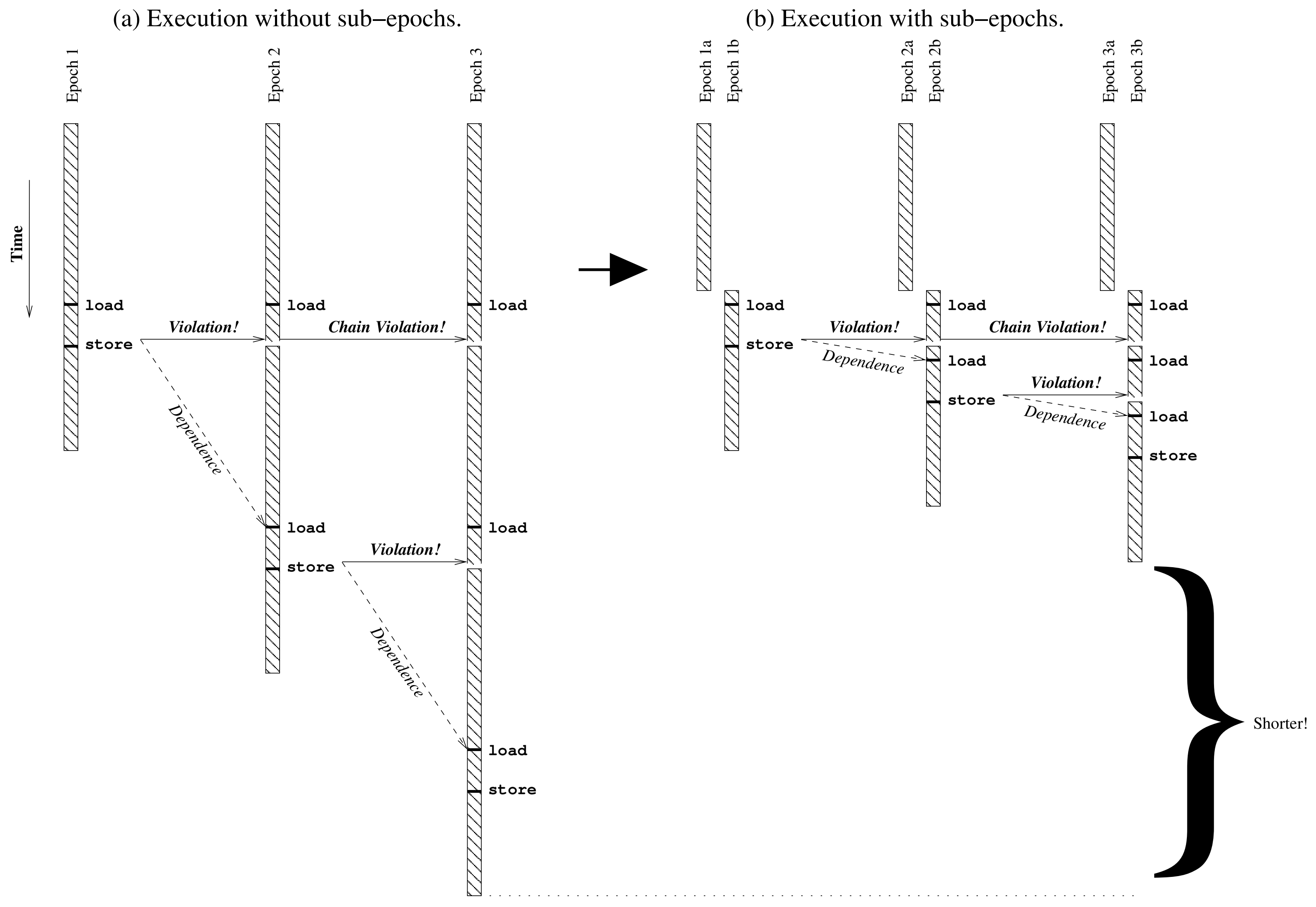


Figure 2.4: Sub-epochs reduce the impact of a dependence late in the epoch.

then more correct execution is rewound. We can limit the amount of correct execution rewound by using *sub-epochs*. A sub-epoch can be viewed as a *checkpoint* of an epoch, or like a nested transaction [41]. Each epoch is divided into multiple sub-epochs. When a violation is detected, it detects which sub-epoch contained the dependent load, and only that sub-epoch (and later sub-epochs) is restarted.

Figure 2.4 shows the effect of dividing each epoch into two sub-epochs. Since the errant load occurs shortly after the second sub-epoch starts, very little correct execution is rewound when the violation is detected. If you compare Figures 2.3(a) and 2.4(b) you see that when using sub-epochs, eliminating the first dependence in Figure 2.3(a) *improves* performance.

If using sub-epochs had no cost, then the best performance would be obtained by starting a new sub-epoch before every load instruction—this would completely avoid rewinding correct execution when a violation occurs. In Chapter 4 we will see that each sub-epoch consumes finite hardware resources: each additional sub-epoch adds state to each cache line and adds complexity to dependence tracking logic. To conserve these resources we adopt a scheme where new sub-epochs 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.

2.3 Life Cycle of an Epoch

To parallelize the loop from Figure 2.1, we need to break the loop up into epochs. The code which does this is shown in Figure 2.5. Note that the loop body is untouched, and we have just added template code to add TLS functionality. In particular, the added template code of Figure 2.5 does the following:

① A new function, `tfork`, is used to create a thread to run each loop iteration as an epoch.[2] 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*.

③ The boundaries of speculative execution are marked with the `become_speculative` and `become_nonspeculative` functions.

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

④ Ensures that the epochs commit their speculative changes in the original sequential program order by passing a *homefree token* from one thread to the next. When a thread possesses the homefree token it is said to be *homefree*, and will no longer be violated by an older epoch.

The execution of our sample loop with these primitives added is illustrated in Figure 2.6(a).

2.4 Moving Code to Avoid Dependences

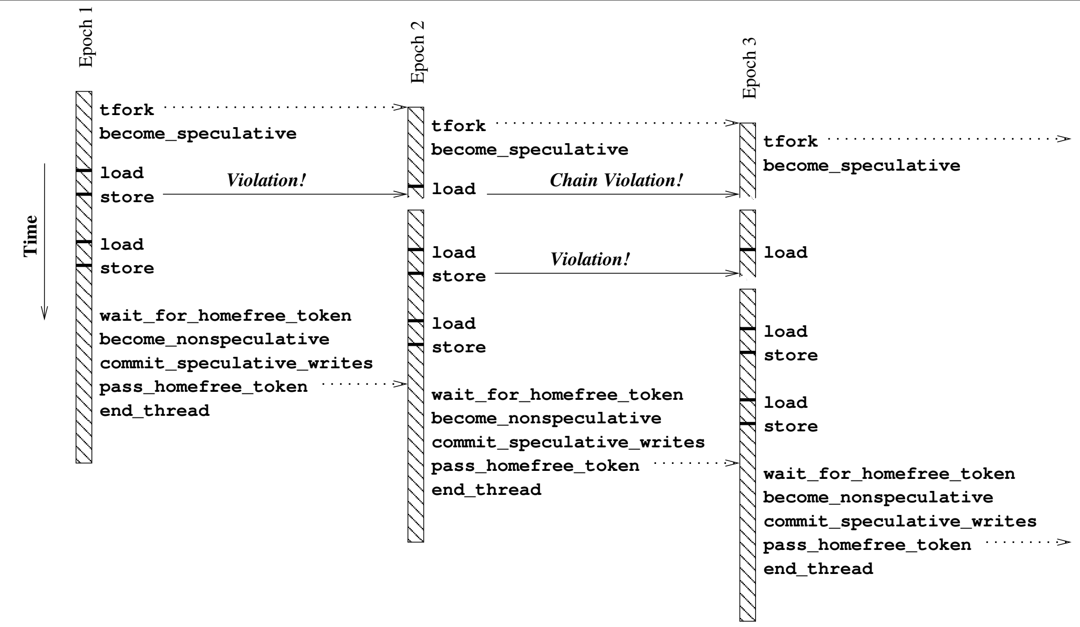
By making these basic TLS primitives (forking epochs, homefree token passing, speculation boundaries) visible to software, we allow the software to be flexible in its use of epochs. In Figure 2.6(b), we show four interesting regions of an epoch's execution where transaction code can be placed. The default location for all of the code in the epoch is in the *speculative region* (❷ in Figure 2.6(b))—code placed in the speculative region executes speculatively, and the TLS mechanism ensures that execution in 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 downwards into the other regions, which are 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 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 *pre-fork region* (❶ in Figure 2.6(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 thesis we assume it does) then the loop index value becomes an argument to the next epoch.

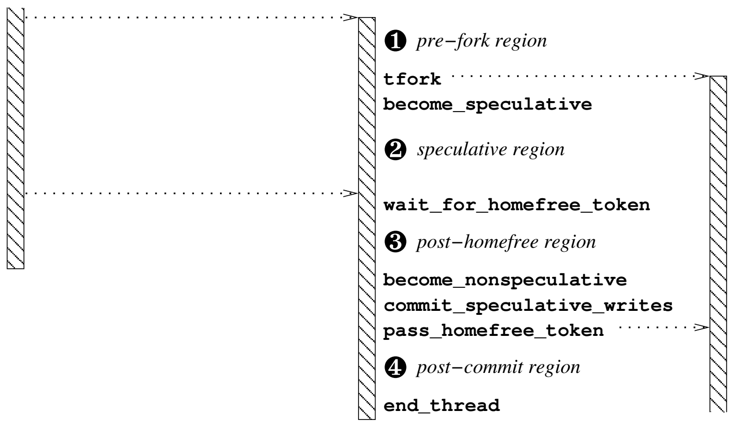
Any code placed in the *post-homefree region* (❸ in Figure 2.6(b)) will not be violated by an earlier thread, since waiting for the homefree token ensures that earlier threads will have committed all of their speculative writes. If some code frequently causes violations, and if that code's execution can be delayed without affecting correctness, then delaying it until the homefree token has arrived (the post-homefree 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

```
/* 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();
    }
  }
}
```

Figure 2.5: Example loop with TLS primitives



(a) TLS primitives in action.



(b) Regions of an epoch's execution.

Figure 2.6: Detailed view of an epoch showing TLS primitives and how they break an epoch into regions of execution.

no code in the epoch depends on the generated numbers, it is safe to delay their generation until the post-homefree region.

If too much code is relocated to the post-homefree 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 *post-commit region* (❹ in Figure 2.6(b)). To delay execution until after the homefree token has been passed the code must be thread safe, since it will be executed non-speculatively in parallel with other threads. An example of code which can be delayed until the post-commit region is calls to `free`. When a transaction frees memory its execution will be unchanged if the `free` is delayed until after the epoch commits.

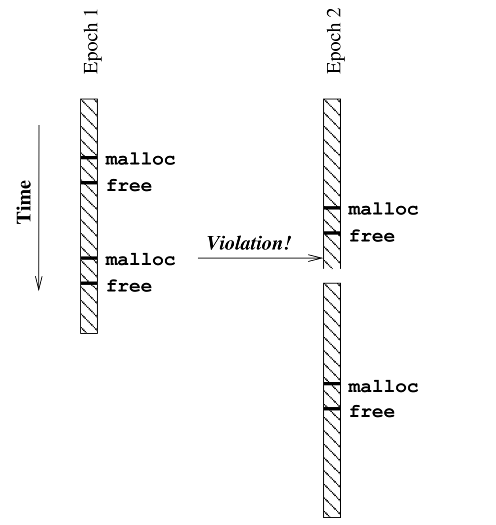
In Chapter 3 we will show numerous examples of how moving code out of the speculative region and into the pre-fork region, post-homefree region and post-commit region can avoid violations when parallelizing the DBMS.

2.5 Avoiding Dependences by Escaping Speculation

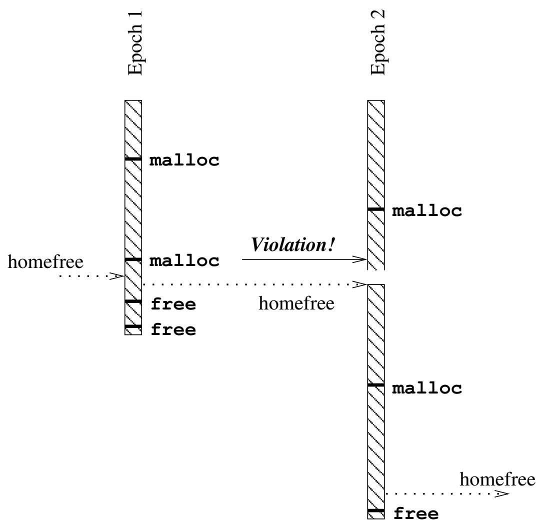
Dependences between epochs can often be easier to understand if you look at the higher-level operations being performed, instead of focusing on the individual loads and stores which cause the dependence. For example, consider two epochs which invoke `malloc` and `free`, as shown in Figure 2.7(a). Both 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` which 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 post-commit 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 post-commit region (Figure 2.7(b)).

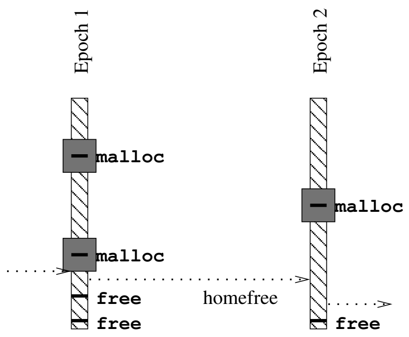
It is not possible to move the call to `malloc` downwards, since the epoch can not proceed until the requested memory is allocated. Instead, we avoid the dependence by *escaping* the speculation mechanism. Fundamentally, to escape speculation we non-speculatively allocate the memory when requested, and *recover* by freeing the memory again if speculation fails. To escape the speculation mechanism, we wrap the `malloc` function with a routine which temporarily disables the speculation mechanism while executing `malloc`. This wrapper must also carefully check the arguments to `malloc` (to avoid ridiculously large or frequent memory allocations, which could cause memory exhaustion), and register a handler which will call `free` on the allocated memory if the epoch is later violated.



(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.

Figure 2.7: Removing dependences between `malloc` and `free`.

```
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;
}
```

Figure 2.8: Wrapper for the `malloc` function which escapes speculation to avoid dependences.

The code wrapper shown in Figure 2.8 implements this modified version of `malloc`. 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 which 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.

③ Temporarily escapes speculation. While speculation is escaped, the epoch is *non-speculative* and hence all reads will observe committed machine state and all writes will be immediately visible to the rest of the system (i.e., no buffering occurs). 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 [38], 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 which satisfies the isolated and undoable properties—in Chapter 3 we will see further examples of escaping speculation in use.

2.6 Inter-transaction Data Dependences

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

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 which 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 [12].[3]

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, which restarts the epoch from the start of the appropriate sub-epoch. 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 non-speculative 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*

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

between the other thread and the epoch. In Chapter 3 we show software transformations which eliminate data dependences between epochs in a single thread. These software transformations also serve to eliminate data dependences *between* threads.

2.7 Chapter Summary

- TLS lets programmers introduce parallelism into a transaction without having to fully understand what data dependences exist in the DBMS's code.
- Sub-epochs allow transactions to *tolerate* backwards data dependences by allowing a violation to discard mis-speculated execution without discarding a large amount of correct execution.
- Moving code in an epoch either upwards or downwards can be used to avoid performance limiting data dependences.
- For certain operations speculation can be *escaped* to avoid performance limiting data dependences.
- 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.

Chapter 3

Applying TLS to Database Transactions

In the previous chapter, we showed techniques for parallelizing programs using TLS. In this chapter we apply TLS to database transactions. We do this by parallelizing the transactions in the TPC-C benchmark [15].

In Section 3.1 we start with the perspective of the transaction programmer. This section shows an implementation of the TPC-C transactions [15] based on the BerkeleyDB DBMS [42], and shows how the transaction programmer can divide the transactions into epochs without detailed knowledge of the internals of the DBMS.

The performance of the resulting epochs is limited by data dependences between them. The majority of these dependences are due to code in the DBMS. This chapter describes the performance bottlenecks we encountered (Section 3.3), and provides a generalized technique for eliminating the data dependences which cause them (Section 3.2).

While in this chapter we evaluate TPC-C transactions running on BerkeleyDB, our techniques can be generalized in two important ways. First, the changes we made were to DBMS data structures and functions which are shared by all transactions, hence the optimizations we describe can be applied to any transaction. Second, we change fundamental primitives used by all database systems (such as latches and locks), hence our techniques are not specific to BerkeleyDB and can be applied to other database systems. Applying our techniques required changing less than 1200 lines out of 180,000 lines of code in the DBMS, and took a graduate student about one month of work. As a result we eliminate 46 to 66% of the latency from three out of five TPC-C transactions on a four CPU system (Section 3.4).

3.1 The Transaction Programmer: Choosing Epoch Boundaries

To apply TLS to database transactions, the transaction programmer (or compiler) must first divide the transaction code into epochs. To understand this process, we worked our way through several examples. We chose to apply TLS to the transactions from the TPC-C benchmark [15]: these transactions represent an important class of workloads (commercial workloads).

Since the TPC-C benchmark specification [62] 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 [42]. The implementation is a straightforward and reasonably efficient implementation of the transactions as specified, and does not contain extensive performance optimizations (the complete transaction code is listed in Appendix A). We did not heavily optimize the transactions since we assume that if TLS is readily available and easy to use then one of the first performance optimizations attempted by the transaction programmer would be to enable TLS.

The transaction programmer has limited knowledge of the internals of the DBMS, and chooses epochs to minimize data dependences which are apparent in the transaction code they are writing. In Section 3.2 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. In the transactions in this thesis the transaction programmer interacts with the DBMS through 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 which 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 sub-epoch as appropriate. To the TLS mechanism database operations look just like any other memory operations. 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 which 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 which occur due to reads and writes to local variables and due to database operations.

3.1.1 NEW ORDER

In Figure 3.1(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 3.1(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 3.1(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 3.4 we will see that this is true.

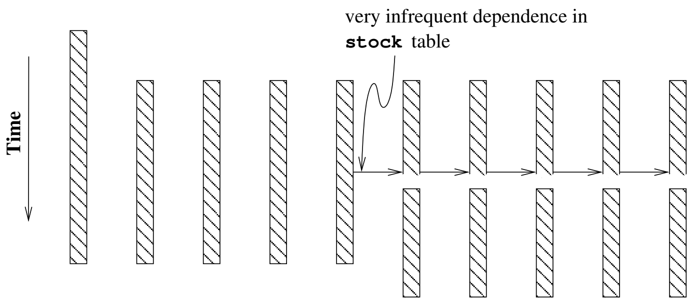
3.1.2 DELIVERY

The DELIVERY transaction (Figure 3.2(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).

```
// 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.

Figure 3.1: The NEW ORDER transaction.

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[1] [31] 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 3.2(b). From this figure one can see that a frequent dependence violation which 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 re-executing epochs grows, since the dependence becomes more and more of a bottleneck. In Section 3.4 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 3.2(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.

3.1.3 STOCK LEVEL

The STOCK LEVEL transaction (Figure 3.3(a)) is a read-only transaction which checks recently ordered items to see if any items in the warehouse have almost run out. This transaction contains one loop which dominates the transaction's execution, 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*—at the start of each epoch the cursor is read, and at the end of each epoch the cursor is incremented. This forms a dependence from the end of each epoch to the start of the next epoch, which completely serializes execution. To avoid this problem we turn the `do-while` loop into a `while` loop, as shown in Figure 3.3(b). This makes the critical path caused by the cursor as short as possible.

The cursor still forms a dependence between the epochs, which causes each epoch to be violated near the start of its execution, as shown in Figure 3.3(b). We could add explicit synchronization to the loop to avoid this violation, but we do not do so since we

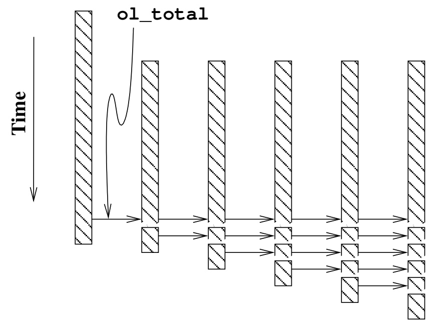
[1]Accumulator variable expansion is used to eliminate a data dependence caused by a variable which accumulates a value, such as a sum in a dot product. In the DELIVERY example each epoch would be given a private copy of the `ol_total` variable to update, and once the loop was complete all of those private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

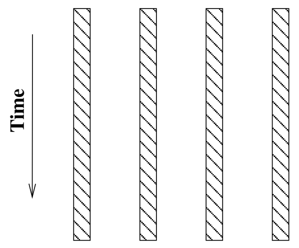
Parallelize this loop

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.

Figure 3.2: The DELIVERY transaction.

```
// 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

(a) Simplified transaction source code.

Figure 3.3: The STOCK LEVEL transaction.

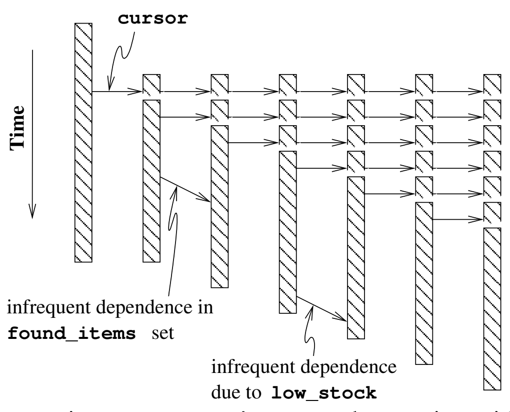
wish to demonstrate the performance gains possible with minimal effort by the transaction programmer. The violations caused by the cursor dependence causes the execution of the rest of the epochs to be somewhat *skewed* in time. This skew ensures that the infrequent dependences on the `found_items` set and `low_stock` variable which occur later in the epoch are executed *in-order*, which prevents them from causing additional violations.

3.1.4 PAYMENT

The PAYMENT transaction (Figure 3.4(a)) is a short transaction which records a payment made by a customer. This transaction contains no loops which 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

```
ol_row = cursor.data();
item_id = ol_row.id;
if(!found_items.contains(item_id)) {
   found_items.insert(item_id);
                                                          Epoch 1
   s_row = stock_table.select(w_id, item_id);
   if(s_row.quantity < threshold) {
      low_stock++;
   }
}
while(cursor = cursor.next()) {
   ol_row = cursor.data();
   item_id = ol_row.id;
   if(!found_items.contains(item_id)) {
      found_items.insert(item_id);
                                                          Epochs 2...n
      s_row = stock_table.select(w_id, item_id);
      if(s_row.quantity < threshold) {
         low_stock++;
      }
   }
}
```

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


cursor
Time
infrequent dependence in found_items set
infrequent dependence due to low_stock

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

Figure 3.3: *Continued.*

threads, as shown in Figure 3.4(b). These two threads cover only 30% of the transaction's execution, which means that they should not offer a large performance gain.

3.1.5 ORDER STATUS

The ORDER STATUS transaction (Figure 3.5(a)) looks up the status of each item ordered by a customer. This transaction contains two loops which cover 38% of the transaction's execution. The majority of the work done in each loop iteration is a cursor lookup and increment, which forms a dependent chain. Both of these loops are dominated by a dependence on the cursor used in them, so even if run in parallel they will execute in a serialized fashion (shown in Figure 3.5(b)).

3.2 The Database System Programmer: Eliminating Dependences in the DBMS

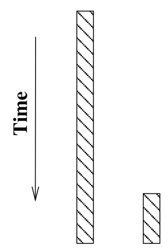
While our transaction-level analysis concludes that TLS parallelization is promising for three of the five TPC-C transactions, the implementation details of query execution algorithms and access methods in the DBMS reveal more potentially performance-limiting data dependences: read/write accesses to 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 2.5. The TLS mechanism ensures that all attempts to fetch and pin a page (`pin_page`) in the buffer pool by one epoch complete before any invocations of `pin_page` in the next epoch begin, due to conflicts in the data structures which maintain LRU information. We prefer to allow `pin_page` operations to complete in any order. An epoch can simply call `pin_page` with speculation 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.*

```
// 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.

Figure 3.4: The PAYMENT transaction.

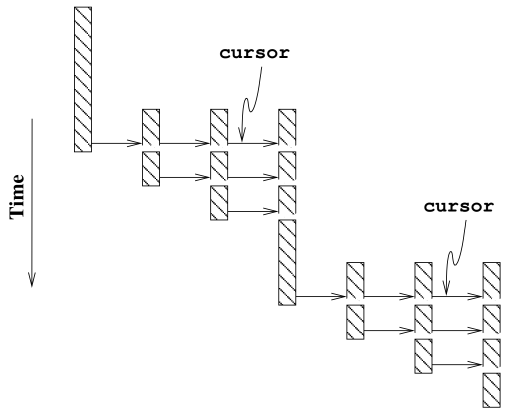
```
// Order Status:  Find the last order by the customer and
// return the status of each item in the order.

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 {
   o_row = cursor.data();
   o_id = o_row.id;
} while(cursor = cursor.next());
i = 0;
cursor = order_line_table.new_cursor(o_id);
do {
   results[i++] = cursor.data();
} while(cursor = cursor.next());
```

Parallelize this loop

Parallelize this loop

(a) Simplified transaction source code.

Figure 3.5: The ORDER STATUS transaction.



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

Figure 3.5: *Continued.*

3. **Postpone operations until the end of the epoch.** Techniques for postponing operations were introduced in Section 2.4. When a log entry is generated, it is assigned a log sequence number and increments a global variable. This log sequence number counter forms a dependence between these two epochs. Our key insight was that an epoch never uses log sequence numbers—it only generates them. We can generate log entries during the execution of the epoch, and assign all of the sequence numbers at the end of the epoch after all previous epochs have completed, and just before committing the epoch (which makes the new log entries visible to the rest of the system). When an operation has no impact on the execution of the epoch, and instead only affects other transactions then it can be delayed until the end of the epoch.

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

3.3 Performance Tuning the DBMS

When we first parallelized the TPC-C transactions we encountered many dependences throughout the DBMS code. Some dependences are easy to eliminate through a local change to the source code: for example, *false sharing* [22] dependences (see Section 3.3.6) can be eliminated by inserting padding in data structures so that independent variables do not share a single cache line. Other data dependences are inherent in the basic design of the database system, such as the creation of log sequence numbers or the locking subsystem. In the following sections we tour the database system's major components, and explain how the database system programmer can eliminate or avoid dependences on the common path in order to increase concurrency for TLS parallelization.

3.3.1 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 which track unused resources are shared between epochs. In the next sections we examine each of these resources and develop strategies for executing them in parallel.

Latches

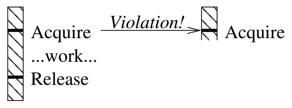
The database system uses latches[2] extensively to protect data structures, and as a building block for locks. Latches are required for correct execution when multiple transactions are executing concurrently, and ensure that only one thread is accessing a given shared data structure at any time. Latches are typically held only briefly—in Section 3.3.1 we discuss *locks*, which offer concurrency control for database entities.

Latches form a dependence between epochs because of how they are implemented: a typical implementation uses a read-test-write cycle on a memory location (which may be implemented as a test-and-set, load-linked/store-conditional, atomic increment, etc.). This read-test-write cycle can cause a data dependence violation between epochs (Figure 3.6(a)).

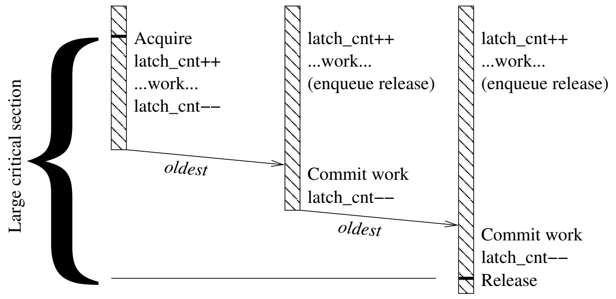
The TLS mechanism already ensures that any data protected by the latch is accessed in a serializable order *within a transaction*, namely the original sequential program order. However, latches do ensure that mutual exclusion is maintained *between transactions*, and TLS does not perform that function. 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

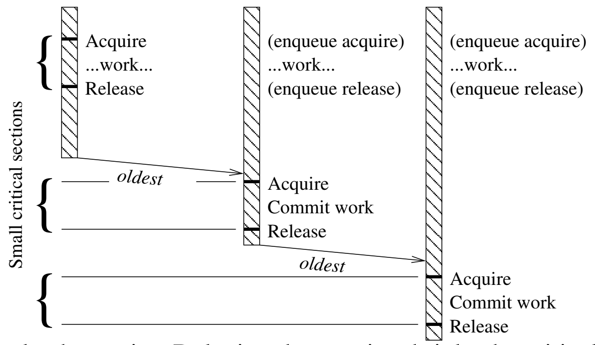
[2]The term *latch* is from the field of databases. A latch is equivalent to a *mutex* in operating systems parlance.



(a) Latch operations create dependences.



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

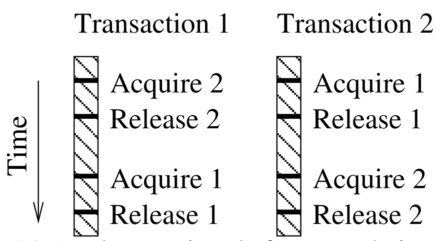


(c) Lazy latch acquire. Delaying the acquire shrinks the critical section.

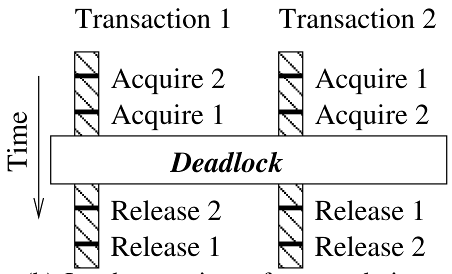
Figure 3.6: Adapting latches for use under TLS execution.

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 2.6(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 *non-speculatively* at the point when it is encountered. This can be implemented by a recursive latch, which counts the number of acquires and releases, and makes the latch available to other transactions only when the count reaches zero. This *aggressive* approach, shown in



(a) Latch operations before re-ordering.



(b) Latch operations after re-ordering.

Figure 3.7: Delaying latch release operations until after a epoch commits can introduce deadlock.

Figure 3.6(b), has a major drawback: since latch releases have been delayed until the end of the epoch, we have increased the overall size of the critical section. In addition, since we have parallel overlap between multiple critical sections in a single transaction, the latch may be held for an extended period of time.

To avoid long critical sections, we can also postpone acquires to the *post-homefree* region of the epoch as shown in Figure 3.6(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.[3]

Both the lazy and aggressive latch schemes have a potential problem: they re-order the latch release operations relative to the latch acquire operations as specified in the original program. If multiple latches are acquired by a single epoch, a deadlock may emerge that is not possible in the sequential execution, as shown in Figure 3.7. 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 [54], such as by sorting the acquire queue by latch address. If avoidance is not possible, we can instead recover from deadlock once detected (perhaps through a time-out) by violating and restarting one of the deadlocked epochs. Forward progress is guaranteed because there is always at least one

[3]This is similar to optimistic concurrency control [29], 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).

epoch (the oldest) which executes non-speculatively. The key insight is that restarting an epoch is much cheaper than restarting the entire transaction since there are many epochs per transaction.

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 3.3.1. When an acquire operation is encountered in speculative code, we cannot simply delay the entire acquire operation until the *post-homefree* region of the epoch, since a handle must be returned. Instead, we return an *indirect* handle, which is a pointer to an empty handle that is filled in at the end of the epoch when the lock acquire is actually performed.

To summarize our scheme so far, at the end of an epoch all of the lock acquires encountered in that epoch are performed, the changes made by the epoch are committed, and then all of the lock releases encountered in the epoch are performed. This scheme will result in correct execution, but holding all of the locks used by an epoch *simultaneously* can be a performance bottleneck in the database, particularly for the locks used for searching B-trees. We avoid this problem by recongizing 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, 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.[4] If latches were labeled as read-only or read/write then this optimization could also be applied to latches in addition

[4]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.

to locks.

Cursor Management

Cursors are data structures used to index into and traverse B-trees. Since they are used quite frequently and their creation is expensive, they are maintained in pre-allocated stacks. Unused cursors are stored in a *free cursor stack*. A dependence between epochs is created when one epoch puts a cursor onto the free cursor stack and the next epoch removes that cursor from the stack, since both operations manipulate the free pointer. Preserving this dependence is not required for correct execution: the second epoch did not need to get the exact same cursor, but instead wanted to get *any* cursor from the free stack. We can eliminate this dependence by partitioning the stack, and hence maintaining a separate stack for each processor. This implies that more cursors will have to be allocated, but that each cursor will only be used by the CPU which allocated it, increasing cache locality and eliminating dependences between epochs. An alternative technique would be to *escape* speculation when allocating cursors, as described in Section 2.5.

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 [8] is one such allocator, which 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 2.5.

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

Buffer Pool Management

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

Conceptually, the buffer pool is very similar to the memory allocator, since it manages memory. However, the buffer pool is different because users explicitly name the memory they want, and different `pin_page` operations can pin the *same page*. Therefore, simply partitioning the page pool between epochs will not suffice. Instead, we exploit the fact that the order in which `pin_page` operations take place *does not matter.* If a speculative epoch fetches the wrong page from disk, we simply must return that page to the free pool. We implement this by executing the `pin_page` function non-speculatively, so that it really does get the page and pin it in a way which is visible to the entire system. If the epoch which called `pin_page` is later violated, we can undo this action by calling `unpin_page`. (This is similar to the compensating transactions used in Sagas [10].)

To execute the `pin_page` function non-speculatively we *escape speculation*, as described in Section 2.5. 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 3.3.1, 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.

3.3.2 The Log

Every time the database is modified the changes are appended to the log. For recovery to work properly (using ARIES [38]) 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 *post-homefree* 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.

3.3.3 B-Trees

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

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

3.3.4 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.

3.3.5 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 which tracks how many cursors are currently in use by a transaction, and ensures that none are in use when the transaction commits. Since this code is solely for debugging a transaction implementation, it can be safely removed once the transaction has been thoroughly tested.

3.3.6 False Sharing

To minimize overhead, the TLS mechanism tracks data dependences at the granularity of a cache-line. However, accesses to different variables which happen to be allocated on the same cache line can cause data dependence violations due to *false sharing*. This problem can be remedied by inserting padding to ensure that variables which are frequently-accessed by different CPUs are not allocated on the same cache line.[5]

3.4 Experimental Results

In this section we evaluate the ease with which a database system programmer can parallelize transactions, and show the resulting performance gains.

3.4.1 Benchmark Infrastructure

Our experimental workload is composed of the five transactions from TPC-C (NEW ORDER, DELIVERY, STOCK LEVEL, PAYMENT and ORDER STATUS).[6] 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

[5]Insertion of padding works for most data structures, but is not appropriate for data structures which mirror disk-resident data, such as B-tree page headers. In this case, changes will have to be made to the B-tree data structure itself (see Section 3.3.3).

[6]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 thesis 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 thesis should not be treated as actual TPM-C results, but instead should be treated as representative transactions.

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*; 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 (100MB) buffer pool.[7]

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; (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, which 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 which 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 3.3 and repeat.

3.4.2 Simulation Infrastructure

We perform our evaluation using a detailed, trace-driven simulation of a chip-multiprocessor composed of 4-way issue, out-of-order, superscalar processors similar to the MIPS R14000 [66], 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 [34] with 16KB, 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 3.1. The TLS mechanism is implemented using the hardware design which is described later in Chapter 4, which includes hardware support for large epochs and 8 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.

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

Table 3.1: 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 Integer	1 cycle
FP Divide	15 cycles
FP Square Root	20 cycles
All Other FP	2 cycles
Branch Prediction	GShare (16KB, 8 history bits)

Memory Parameters	
Cache Line Size	32B
Instruction Cache	32KB, 4-way set-assoc
Data Cache	32KB, 4-way set-assoc,2 banks
Unified Secondary Cache	2MB, 4-way set-assoc, 4 banks
Speculative Victim Cache	64 entry
Miss Handlers	128 for data, 2 for insts
Crossbar Interconnect	8B per cycle per bank
Minimum Miss Latency to Secondary Cache	10 cycles
Minimum Miss Latency to Local Memory	75 cycles
Main Memory Bandwidth	1 access per 20 cycles

The simulator used to generate these results is a trace driven simulator, which means 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 mis-speculation (both due to TLS and due to branch mis-speculations) are simulated by executing the correct path twice, which may slightly underestimate instruction cache misses. Recovery code (such as code executed to undo mis-speculated `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

Table 3.2: Benchmark statistics.

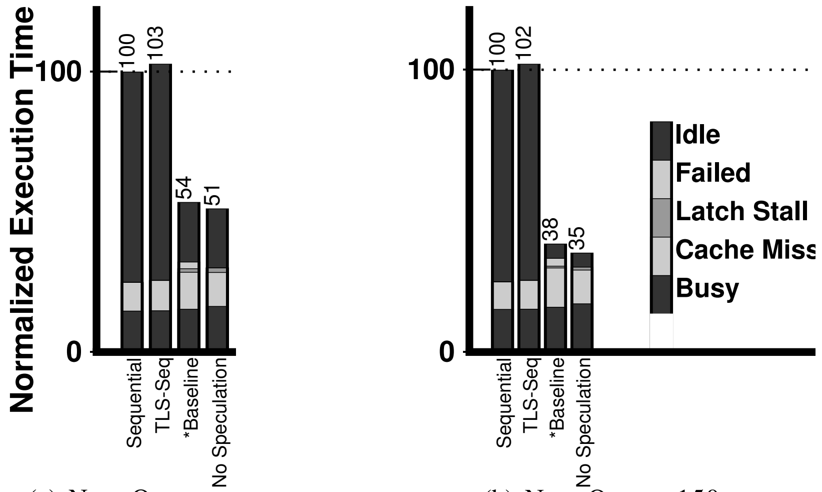
Benchmark	Sequential Exec. Time (Mcycles)	Coverage	Average Epoch Stats: Size (Dyn. Instrs.)	Spec. Insts. per Epoch	Threads per Transaction
NEW ORDER	62	78%	62k	35k	9.7
NEW ORDER 150	509	94%	61k	35k	99.6
DELIVERY	374	63%	33k	20k	10.0
DELIVERY OUTER	374	99%	490k	327k	10.0
STOCK LEVEL	253	98%	17k	10k	191.7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

of delaying the violation until speculation resumes.

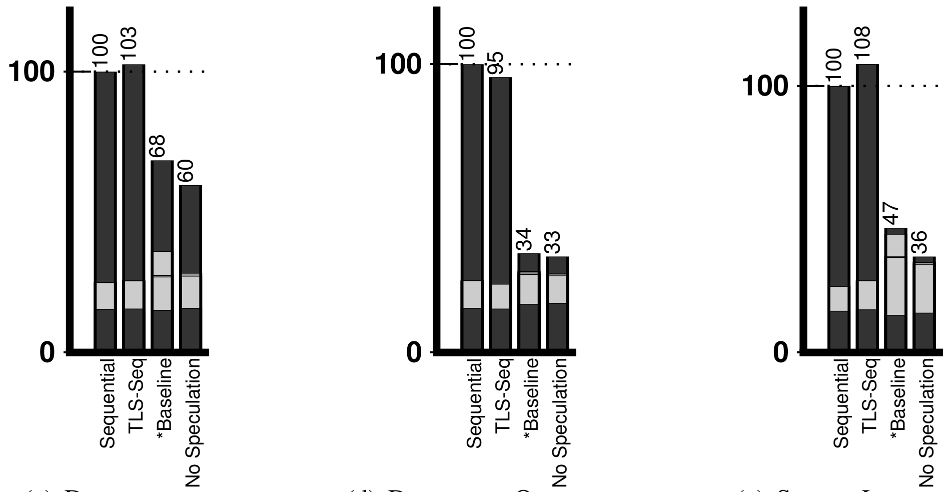
3.4.3 High Level Benchmark Characterization

We start by characterizing the benchmarks, so we can better understand them. As a starting point for comparison, we run our original sequential benchmark, which shows the execution time with no TLS instructions or any other software transformations running on one CPU of the machine (which is configured with 4 CPUs, cache line replication, and sub-epoch support enabled). This SEQUENTIAL experiment takes between 17 and 509 million cycles (Table 3.2) to execute, but this time is normalized to 100 in Figure 3.8. (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 which 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 Figure 3.8 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 (described further in Chapter 4, with 4 CPUs, 8 sub-epochs per epoch, 5000 speculative instructions per sub-epoch) 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 3.1 the PAYMENT contains no loops worth parallelizing, and the threads we chose were limited by a dependence in the locking

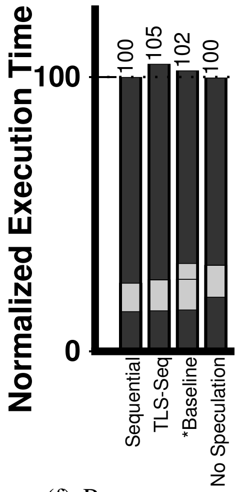


(a) NEW ORDER (b) NEW ORDER 150

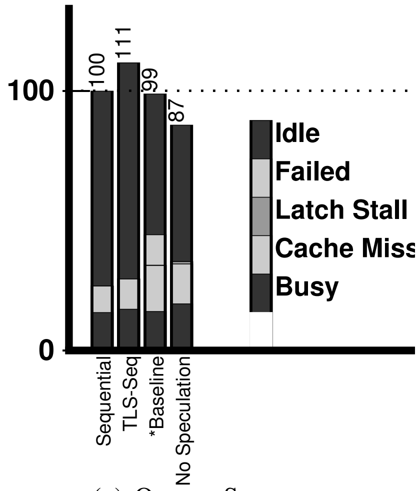


(c) DELIVERY (d) DELIVERY OUTER (e) STOCK LEVEL

Figure 3.8: Overall performance of optimized benchmark on 4 CPUs.



(f) PAYMENT



(g) 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 thesis.
No Speculation	Upper bound—modified hardware to treat all speculative writes as non-speculative.

Figure 3.8: *Continued.*

code which did not affect the other transactions; the main loop in ORDER STATUS contains an unavoidable dependence on a cursor operation.

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 non-speculative (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 non-parallelized 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

Table 3.3: Explanation of graph breakdown.

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

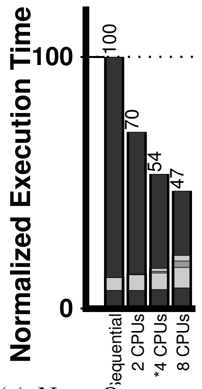
the ORDER LINE table. The STOCK LEVEL transaction is limited by dependences on the cursor used to scan the ORDER LINE table.

3.4.4 Scaling Intra-Transaction Parallelism

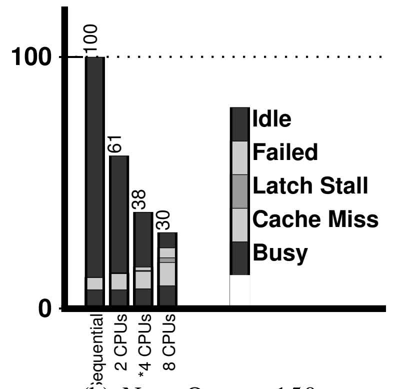
In Figure 3.9 we see the performance of the fully optimized transactions as the number of CPUs is varied. The SEQUENTIAL bar represents the unmodified benchmark running on a single core of an 8 core chip multiprocessor, while the 2 CPU, 4 CPU and 8 CPU bars represent the execution of full TLS-optimized executables running on 2, 4 and 8 CPUs. Large improvements in transaction latency can be obtained by using 2 or 4 CPUs, although the additional benefits of using 8 CPUs are small.

Each bar is divideded into subdivisions which show what each CPU is doing for each cycle of execution. The subdivisions are explained in Table 3.3. In Figure 3.9 we have normalized all bars to the 8 CPU case so that the subdivisions of each bar can be directly compared. This means that the SEQUENTIAL breakdown shows one CPU executing and seven CPUs idling, the 2 CPU breakdown shows two CPUs executing and six CPUs idling, etc.

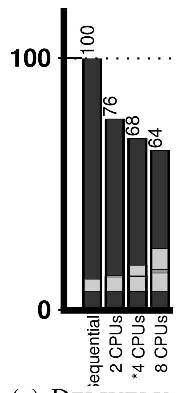
The NEW ORDER, NEW ORDER 150 and DELIVERY OUTER bars show that very little time was spent on failed speculation—this means that our performance tuning was successful at eliminating performance-critical data dependences for those transactions. The DELIVERY transaction has some failed speculation due to 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 exe-



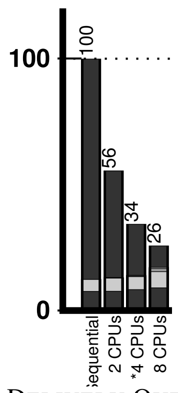
(a) NEW ORDER



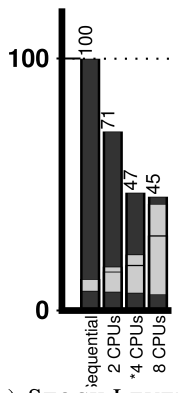
(b) NEW ORDER 150



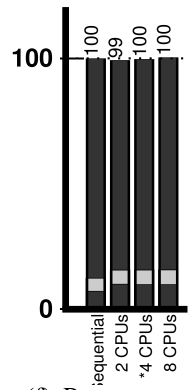
(c) DELIVERY



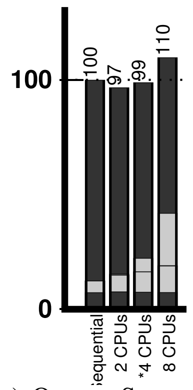
(d) DELIVERY OUTER



(e) STOCK LEVEL



(f) PAYMENT



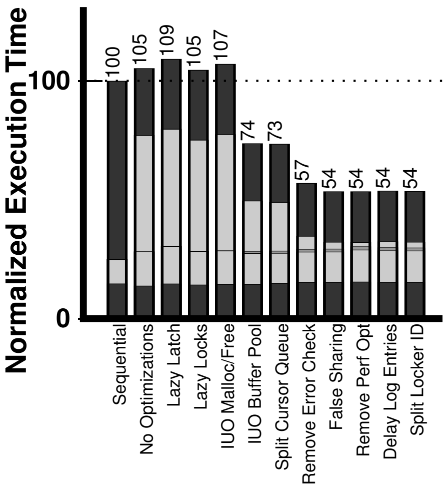
(g) ORDER STATUS

Figure 3.9: Performance of optimized benchmark while varying the number of CPUs.

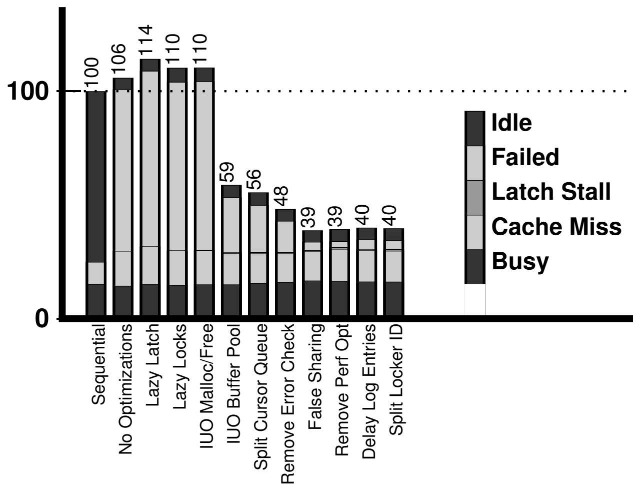
cution of the transaction over more CPUs decreases cache locality, since the execution is partitioned over more level 1 caches. We also see a much larger increase in the number of cache misses for the STOCK LEVEL transaction—a large amount of cache state can be invalidated when speculation fails, leading 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, 2 CPU and 4 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 which contain between 5 and 15 items, which means that on average each transaction will have only 10 epochs—this means that as we execute the last epochs in the loop load imbalance will leave CPUs idling. The effects of all three of these issues are magnified as more CPUs are added. To 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 (which is 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.

Figure 3.9 shows a performance trade-off: 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 intra-transaction parallelism is that it can be enabled or disabled at any time, and the number of CPUs can be dynamically tuned. The database system's scheduler can dynamically increase the number of CPUs available to a transaction if CPUs are idling, or to speed up a transaction which holds heavily contended locks. If many epochs are being violated, and thus the intra-transaction parallelism is providing little performance benefit, then the scheduler could reduce the number of CPUs available to the transaction. If the transaction compiler simply emitted a TLS parallel version of *all* loops in transactions then the scheduler could use sampling to choose loops to parallelize: the scheduler could periodically enable TLS for loops which are not already running in parallel, and periodically disable TLS for loops which are running in parallel. If the change improves performance then the scheduler can make it permanent.



(a) NEW ORDER



(b) NEW ORDER 150

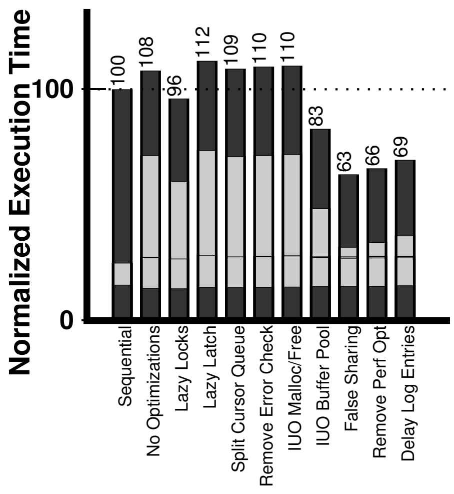
Figure 3.10: Performance impact on the TPC-C transactions of adding each optimization one-by-one on a four CPU machine.

3.4.5 Impact of Each Optimization

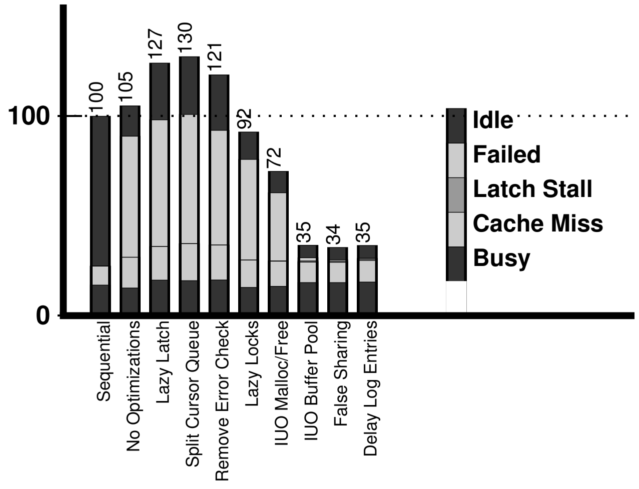
In Figure 3.10 we see the results of the optimization process for the benchmarks which sped up on a four CPU system. In this experiment the breakdown of the bars is normalized to a four CPU system, and so $\frac{3}{4}$ of the SEQUENTIAL bars is *Idle*, since three of the four CPUs are idling during the 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%.

Consider the NEW ORDER transaction in Figure 3.10(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 3.3.1. This optimization fixes the first performance bottleneck, and exposes the next bottleneck which 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 2.3)—hence more execution has to be rewound.

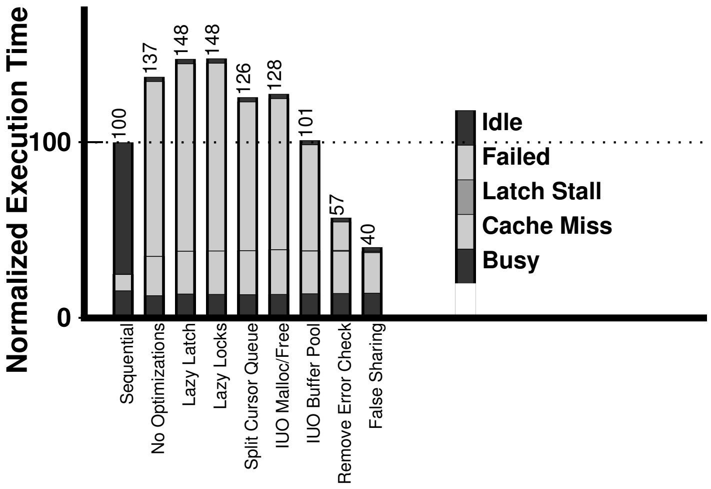
Once we have eliminated latches as a bottleneck in NEW ORDER, the next bottleneck exposed is in the locking subsystem. We remove the lock bottleneck by implementing lazy locks from Section 3.3.1. We continue to remove the bottlenecks one by one: applying the



(c) DELIVERY



(d) DELIVERY OUTER



(e) STOCK LEVEL

Figure 3.10: *Continued.*

code template from Figure 2.8 to `db_malloc` and the `pin_page` operation, parallelizing the free cursor pool, removing dependence causing error checks (Section 3.3.5), adding padding to avoid violations due to false sharing (Section 3.3.6), removing the "last page referenced" pointer from the B-tree search code (Section 3.3.3), delaying the generation of log entries until epochs are ready to commit (Section 3.3.2), and parallelizing the assignment of locker ids.

It is tempting to look at Figure 3.10(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 3.10 implies that the iterative optimization process which 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.

Figure 3.10(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 3.10(c), (d) and (e) 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 3.10. 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.

3.5 Chapter Summary

- To parallelize a transaction with TLS the transaction programmer only has to choose epoch boundaries.
- Choosing epoch boundaries can be as simple as choosing an appropriate loop nest.
- The database system programmer optimizes for TLS through an iterative *performance tuning* process.
- Localized changes to the DBMS can remove performance limiting data dependences.
- 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 46–66% by exploiting TLS on a chip multiprocessor with four CPU cores, or by 29–44% with two cores.

Chapter 4

Hardware Support for Large, Dependent Epochs

In the previous chapters we saw that applying TLS to database transactions is quite worthwhile, and TLS is able to reduce transaction latency by a factor of two on a simulated 4-CPU machine. This success required us to overcome some new challenges that did not arise in previous studies of smaller programs such as the SPEC [56] benchmarks. In particular, after breaking up the TPC-C transactions based upon natural sources of parallelism in the SQL code, the resulting speculative epochs were much larger than in previous studies (the majority of the TPC-C transactions that we studied had more than 50,000 dynamic instructions per epoch), and there were far more inter-epoch data dependences (due to internal database structures—not the SQL code itself) than in previous studies. As a result, we observed no speedup on a conventional TLS architecture.

There were three aspects to overcoming these challenges so that we could go from no speedup to significant (e.g., twofold) speedup. First, the baseline TLS hardware needed to support large epochs and aggressive value forwarding between epochs. Second, the programmer needs to be able to identify data dependence bottlenecks and eliminate them through a combination of data dependence profiling feedback and the ability to temporarily escape speculation. Finally, to avoid wasting useful work when the remaining dependences still cause speculation to fail, we propose a mechanism for *tolerating* failed speculation by using lightweight checkpoints to roll a epoch back to an intermediate point before speculation failed.

4.1 Hardware Support for Large Epochs

TLS allows us to break a program's sequential execution into parallel speculative epochs, and ensures that *data dependences* between the newly created epochs are preserved. Any read-after-write dependence between epochs which is *violated* must be *detected*, and *corrected* by re-starting the offending epoch. Hardware support for TLS makes the detection of violations and the restarting of epochs inexpensive [18, 47, 55, 60].

Our database benchmarks stress TLS hardware support in new ways which have not been previously studied, since the epochs are necessarily so much larger. Previous work has studied epochs with various size ranges, including 3.9–957.8 dynamic instructions [65], 140–7735 dynamic instructions [46], 30.8–2,252.7 dynamic instructions [61], and up to 3,900–103,300 dynamic instructions [11]. The epochs studied in this thesis are quite large, with 7,574–489,877 dynamic instructions. These larger epochs present two challenges. First, more speculative state has to be stored for each epoch (from 3KB to 35KB, before storing multiple versions of cache lines for sub-epochs). Most existing approaches to TLS hardware support cannot buffer this large amount of speculative state. Second, these larger epochs have many data dependences between them which cannot be easily synchronized and forwarded by the compiler, since they appear deep within the database system in very complex and varied code paths. This problem is exacerbated by the fact that the database system is typically compiled separately from the database transactions, meaning that the compiler cannot easily use transaction specific knowledge when compiling the database system. This makes runtime techniques for tolerating data dependences between epochs attractive.

Previous work on TLS assumes that the speculative state of a epoch fits in either speculative buffers [18, 55] or in the L1 cache [6, 14, 60]. With the large amount of speculative state per epoch used by our database benchmarks we find that we suffer from conflict misses in the L1 cache (which cause speculation to fail). Increasing the associativity does not necessarily solve the problem, since even fully associative L1 caches may not be large enough to avoid capacity misses.

Two prior approaches address the problem of cache overflow: Prvulovic *et. al.* proposed a technique which allows speculative state to overflow into main memory [47]; and Cintra *et. al.* proposed a hierarchical TLS implementation which allows the oldest epoch in a CMP to buffer speculative state in the L2 cache (while requiring that the earlier epochs running on a CMP be restricted to the L1 cache) [6]. In this thesis, we propose a similar hierarchical implementation, with one important difference from Cintra's scheme: it allows *all* epochs to store their speculative state in the larger L2 caches. With this support (i) all epochs can take advantage of the large size of the L2 cache, (ii) epochs can aggressively propagate updates to other more recent epochs, and (iii) we can more easily implement

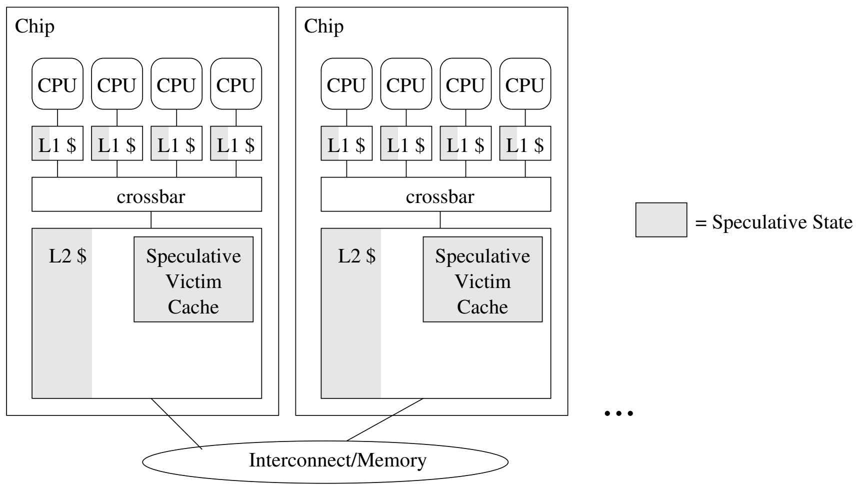


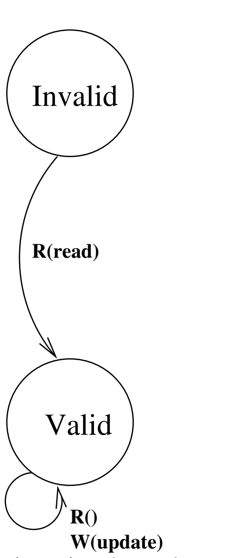
Figure 4.1: An overview of the CMP architecture that we target, and how it is extended to support TLS.

sub-epochs, described later in Section 4.3.

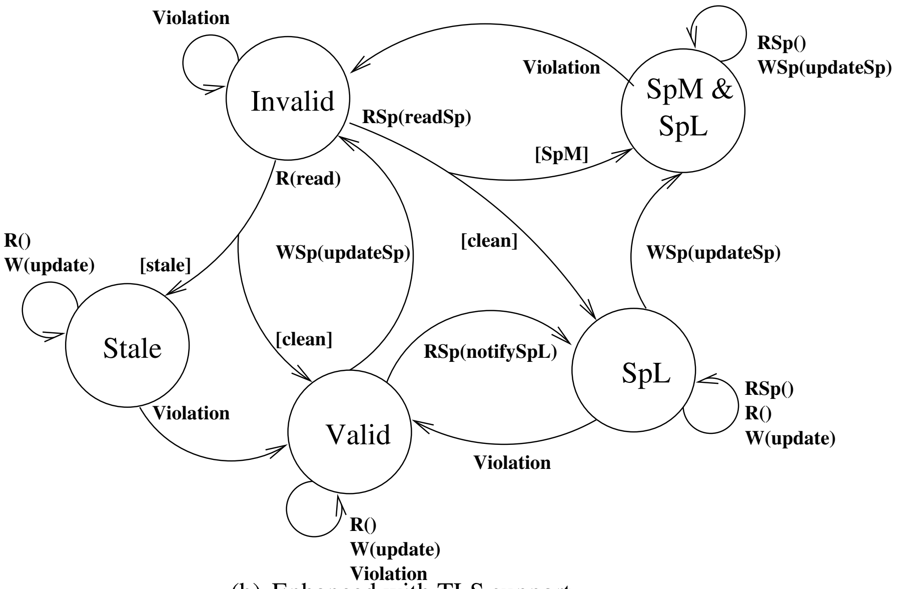
4.1.1 A Protocol for Two-Level Speculative State Tracking and Buffering

In this section we describe the underlying chip multiprocessor (CMP) architecture and how we extend it to handle large epochs. We assume a CMP where each core has a private L1 cache, and multiple cores share a single chip-wide L2 cache (Figure 4.1). For simplicity, in this thesis each CPU executes a single epoch. We buffer speculative state in the caches, detecting violations at a cache line granularity. Both the L1 and L2 caches maintain speculative state: each L1 cache buffers cache lines that have been speculatively read or modified by the epoch executing on the corresponding CPU, while the L2 caches maintain inclusion, and buffer copies of all speculative cache lines that are cached in any L1 on that same chip. Detection of violations between epochs running on *the same chip* is performed within the L2 cache through the two-level protocol described below. For detection of violations between epochs running on *different chips*, we assume support for an existing scheme for distributed TLS cache coherence, such as the STAMPede protocol [60].[1]

[1]In this thesis, all experiments are within the boundaries of a single chip; the amount of communication required to aggressively propagate all speculative stores between chips is presumed to be too costly.



(a) Basic write-through protocol.



(b) Enhanced with TLS support.

Figure 4.2: L1 cache line state transition diagram. Any transitions not shown (for example, action **R** for a line in the state ***SpM***) is due to an impossible action: the tag match will fail, and the transition will be treated as a miss. Line states are explained in Table 4.1. Edges are labeled with the processor action (Table 4.2), followed by the message to the L2 cache in round brackets (Table 4.3). Square brackets show responses from the L2 cache (Table 4.4).

L1 Cache State

Our design has the following two goals: (i) to reduce the number of dependence violations, by aggressively propagating store values between epochs; (ii) to reduce the amount of time wasted on failed speculation, by ensuring that any dependence violation is detected promptly. In our two-level approach, the L2 cache is responsible for detecting dependence violations, and must therefore be informed of loads and stores from all epochs.

In Figure 4.2 we illustrate how to take a simple write-through L1 cache line state transition diagram and enhance it for use with our shared L2 cache design. In the following discussion we explain the new states and the functionality they offer.

To track the first speculative load of an epoch we add the *speculatively loaded* (*SpL*) bit to each cache line in the L1 cache. The first time the processor tries to speculatively

Table 4.1: Explanation of cache line states.

Invalid	Invalid line, contains no useful data.
Valid	Line contains data which mirrors committed memory state.
Stale	Line contains data which mirrors committed memory state, but there *may* exist a more speculative version generated by the L1 cache's CPU.
SpL	Line is valid, but has been speculatively loaded by the L1 cache's CPU.
SpM & SpL	Line has been has been speculatively loaded by the L1 cache's CPU, and the L2 cache returned a line which was speculatively modified by a earlier epoch.

Table 4.2: Explanation of actions.

R	Processor read a memory location.
W	Processor wrote a memory location.
RSp	Processor read a memory location while executing speculatively.
WSp	Processor wrote a memory location while executing speculatively.
Violation	Violation detected (action generated by L2 cache).

Table 4.3: Explanation of messages from L1 to L2 cache.

read	Read memory location, L1 cache miss.
readSp	Read memory location while executing speculatively, L1 cache miss.
update	Wrote to memory location.
updateSp	Wrote to memory location while executing speculatively.
notifySpL	Read memory location while executing speculatively, L1 cache hit.

Table 4.4: Explanation of responses from L2 cache to L1 cache.

clean	Requested line *has not* been speculatively modified by any earlier epoch.
stale	Requested line *has* been speculatively modified by an earlier epoch, but returning non-speculative (committed) version of the line.
SpM	Requested line *has* been speculatively modified by an earlier epoch, returning speculative version of the line.

load a line this bit will be set, and if the load hits in the L1 cache a *notify speculatively loaded* (*notifySpL*) message will be sent to the L2 cache, informing it that a speculative load has occurred. If the processor tries to speculatively load a line which is not present in the L1 cache, it will trigger a *speculative miss* (*readSp*) to the L2 cache.

The *SpL* bit acts as a filter—it ensures that the L2 cache is not notified multiple times of a line being speculatively loaded. The L2 cache learns of speculative loads through the *notifySpL* message and *readSp* request. The *readSp* request is blocking, since the load which triggers it can not complete until the cache miss is serviced. The *readSp* request returns the most up-to-date replica of the cache line. The *notifySpL* message is non-blocking, so the load which triggers it can complete immediately. The purpose of the *notifySpL* message is to allow the L2 cache to detect potential violations—to avoid race conditions which would result in not detecting a violation, all *notifySpL* messages must be processed by the L2 cache before an epoch can commit, and any *notifySpL* messages in transit from the L1 to the L2 cache must be compared against invalidations being sent from the L2 to the L1 cache (this is similar to how a store buffer must check for invalidations). An alternative to using the *notifySpL* message is to instead always use a *readSp* message for the first speculative load of an epoch. This effectively makes each epoch start with a cold L1 cache—the *notifySpL* message is a performance optimization, based on the assumption that the L1 cache will rarely contain out-of-date replicas of cache lines (if the L2 cache receives a *notifySpL* message for a line which has been speculatively modified by an earlier epoch, then this will generate a violation for the loading epoch). In Section 4.4.2 we evaluate the performance impact of this optimization.

Detecting the *last* store to a memory location by an epoch is more challenging. Although we do not evaluate it here, a sophisticated design could combine a write-back cache with a *last-touch predictor* [30] to notify the L2 of only the *last* store performed by a epoch. However, for now we take the more conservative approach of making the L1 caches write-through, ensuring that store values are aggressively propagated to the L2 where dependent epochs may load those values to avoid dependence violations. Each L1 cache line also has a *speculatively-modified bit* (*SpM*) which is set on the first speculative store, so that it can be flash-invalidated if the epoch violates a dependence (rather than relying on an invalidation from the L2 cache).

In our design when the L1 cache holds a line which is marked speculative we assume it holds the most up-to-date replica of the line (containing all changes made by older epochs). Since the write merging used to generate the most up-to-date replica is performed in the L2 cache, the L1 cache can not transition a *Valid* line into a *SpM* line without querying the L2 cache. Because of this, in Figure 4.2 you will see that a speculative write to a *Valid* line invalidates the line so that any loads to that line retrieve the correct speculative replica from the L2 cache.

When an epoch commits the L1 cache can simply clear all of the *SpM* and *SpL* bits, since none of the state associated with the committing epoch or any earlier epoch is speculative. If multiple epochs share a single L1 cache then the *SpM* and *SpL* bits can be replicated, one per epoch, as is done for the shared cache TLS protocol proposed in prior work [57].

The *Stale* state in Figure 4.2 is used for temporarily escaping speculation, and is discussed further in Section 4.2.2.

L2 Cache State

The L2 cache buffers speculative state and tracks data dependences between epochs using the same techniques as used by the TLS protocol proposed in prior work [57]. Instead of observing each load and store from a CPU the L2 cache observes *read*, *readSp*, *notifySpL*, *update* and *updateSp* messages from the L1 caches. Each message from an L1 cache is tagged with an epoch number, and these numbers are mapped onto the *epoch contexts* in the L2 cache. Each epoch context represents the state of a running epoch. Each cache line has a *SpM* and *SpL* bit per epoch context. Dependences between epochs sharing the same L2 cache are detected when an update is processed, by checking for *SpL* bits set by later epochs. Dependences between epochs running on different L2 caches are tracked using the extended cache coherence proposed in prior work [57].

With the large epochs that are found in database transactions we need to tolerate storing large amounts of speculative state in the L2 cache from multiple epochs. Often there are two *conflicting* versions of a line that need to be stored—for example, if a line is modified by different epochs then each modification must be tracked separately so that violations can efficiently undo the work from a later epoch without undoing the work of an earlier epoch. When conflicting versions of a line must be stored we *replicate* the cache line and maintain two versions. Maintaining multiple versions of cache lines has been studied previously in the Multiscalar project [14]; we present our replication scheme in detail in

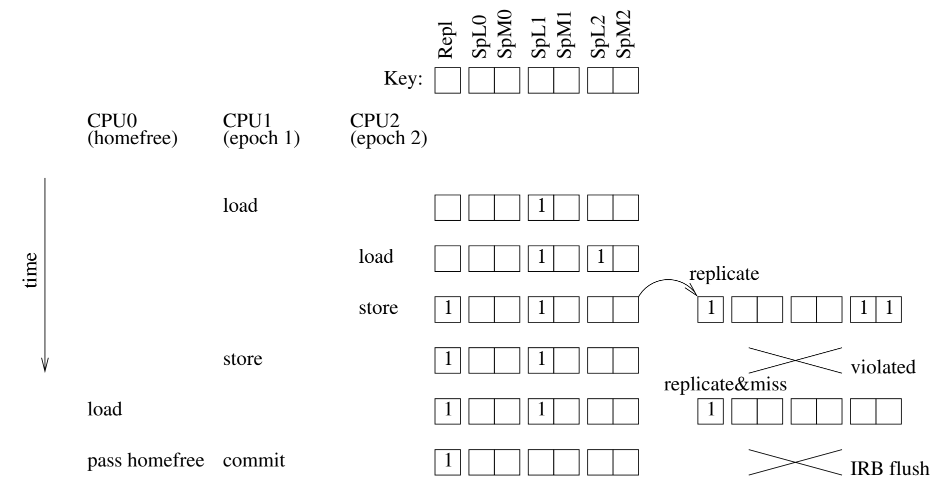


Figure 4.3: Step-by-step example demonstrating cache line replication for a single cache line. Cache line replication avoids violations when a *replica conflict* occurs.

the next section to illustrate an alternate implementation, and since we build upon our replication scheme to implement sub-epochs in Section 4.3.

If a speculative line is evicted from the cache then we need to continue tracking the line's speculative state. With large epochs and cache line replication this becomes more of a problem than it was in the past: we use a *speculative victim cache* [26, 58] to hold evicted speculative lines and track violations caused by these lines. We have found that a small victim cache is sufficient to hold our overflow, but if more space is required we could use a memory-based overflow area such as the scheme proposed by Prvulovic [47] to store the overflowed speculative state.

Cache Line Replication

In a traditional cache design, each address in memory maps to a unique cache set, and tag lookup leads you to a unique cache line. When we have several epochs storing their speculative state in the same cache there can be *replica conflicts*: a replica conflict is when two epochs need to keep different versions of the cache line to make forward progress. There are three cases where a replica conflict can arise. The first class of replica conflict is if an epoch loads from a cache line and a more speculative epoch has speculatively modified that cache line. In this case we do not want the load to observe the more speculative changes to the cache line, since they are from an epoch which occurs later in the original sequential program order. The second class of replica conflict is if an epoch stores to a cache line that any other epoch has speculatively modified. Since we want to be able to commit speculative changes to memory one epoch at a time, we cannot mix the stores from two epochs together. The third class of replica conflict is if an epoch stores to a cache line that any earlier epoch has speculatively loaded. The problem is that a cache line contains both speculative state and speculative meta-state (the *SpM* and *SpL* bits). In this case we want to be able to quickly completely discard the cache line (the speculative state) if the storing epoch is later violated, but we do not want to discard the *SpL* bits (the speculative meta-state) if they are set. To avoid this problem we treat it as a replica conflict.

What can we do if a replica conflict arises? Every replica conflict involves two epochs: if the replica conflict arises when a store is performed by the later epoch, then the later epoch can be stalled until the earlier epoch commits; if the replica conflict arises when a load or store is performed by the earlier epoch then it can be resolved by violating the later epoch. One approach is to stall an epoch until it is homefree when a replica conflict is detected, or violate an epoch when a replica conflict is detected. Both of these approaches hurt performance severely if replica conflicts happen frequently. Another approach is to replicate cache lines—when a replica conflict arises make a fresh copy of the conflicting cache line and use the copy instead (Figure 4.3, Appendix B). As epochs commit these replicas are reunited into a single line.

If there exist multiple replicas of a cache line then any access to the cache has to decide *which* of those replicas to use. The correct answer is to use the *most recent* replica with respect to the current epoch. If epoch e is accessing the cache, then it wants to use replica which was last speculatively modified by epoch e, or if that does not exist then epoch $e - 1$, or if that does not exist then epoch $e - 2$, etc. If the current epoch has not speculatively modified the line *and* no prior epoch has speculatively modified the line then the *clean replica* is used. The clean replica is the replica of the cache line which contains no speculative modifications. Note that if the clean replica is not present in the cache then it can always be retrieved from the next level of the memory hierarchy via a cache miss.

Because the most recent replica with respect to the current epoch must be located, cache lookup for a replicated line may be slightly more complex than a normal cache lookup. To limit the impact of this to lines with replicas, we add a *replica* bit to each cache line which indicates that a replica of the line *may* exist, which is set when a replica of a line is created.

Replicas are always created from the most recent replica. A copy of the source line is made, and the *SpM*, *SpL* and directory bits (which indicate which L1 caches above the L2 cache may have copies of the line) are copied as well. The *SpM*, *SpL* and directory bits representing epochs older than the current epoch are cleared on the newly created line, while the bits representing the current epoch and newer epochs are cleared on the source line. This way the responsibility for tracking speculative state is divided so that older state resides on the source line, and the newly created replica tracks state associated with the current and later epochs.

The existence of replicas slightly complicates store operations. When a store is performed, the changes must be propagated to the newer replicas of the cache line. This means that a store has to write to the most recent replica and also to any newer replicas if they have not already overwritten the same word in the line. The fine-grained *SpM* bits (used for write merging between L2 caches) specify which words in a cache line have been speculatively written to, and they can be used to determine which (if any) newer replicas need to be updated.[2]

When an epoch commits all speculatively modified cache lines associated with the committing epoch are transformed into dirty cache lines, which become clean replicas (since they no longer hold speculative modifications). There may only be one clean replica of a cache line in the cache at a time, we ensure this by having the commit operation first invalidate any clean replicas which are made obsolete by the commit operation. The lines which need to be invalidated are tracked through the *invalidation required buffer* (IRB), which operates in a similar fashion to the ORB [57]. There is one IRB per epoch context. An IRB entry is simply a cache tag, and says "the cache line associated with this address and epoch may have an older replica." When a replica is created it may generate up to two IRB entries. If there exists any replica with state older than the newly created replica's state then an entry is added to the newly created replica's IRB. If there exists any replicas with state newer than the newly created replica's state then an entry is added to the IRB of the oldest of the newer replicas. When an epoch is violated the IRB for that epoch is cleared. When an epoch commits, the cache first invalidate any clean replicas named by IRB entries, then clears the *SpM* and *SpL* bits associated with the committing epoch.

With multiple versions of a single cache line in the L2 cache, it is easy to believe that tracking which L1 cache has which version of the cache line, and keeping the L1 caches up to date is made more complex. This is not the case. Each version in the L2 cache has directory bits which track which L1 caches may have a replica of the line (just like in a cache without replication), and when a replica is updated or invalidated an invalidation is sent to the appropriate L1 caches. In effect, properly maintaining inclusion ensures that the L1 caches are always consistent with the L2 cache.

[2]Alternatively, speculatively modified cache lines can be merged with the clean replica at commit time. This is much harder to implement, since the clean replica can be evicted from the cache at any time if the cache is short on space. This means that a committing epoch may suffer cache misses to bring those clean replicas back into the cache for merging. Another alternative would be to pin the clean replicas of cache lines in the cache until all replicas of a line are merged, but this wastes cache space.

Speculative Victim Cache

One potential problem with storing speculative state in the cache is that cache lines with speculative state cannot be easily evicted from the cache, as this would result in a loss of information crucial to speculation. Replication creates even more speculative cache lines in each cache set (since each line may have multiple replicas), and this increases cache pressure. One way of dealing with overflowing cache sets is to either suspend the execution of an epoch to avoid the eviction or to violate an epoch if its state is evicted. To avoid costly violations and/or stalls, we add a speculative victim cache to the L2 cache. A speculative victim cache is just like a normal victim cache [26], with the addition of mechanisms to:

- check contained lines for violations whenever a write is performed or an external invalidation is received;
- merge writes into newer replicas, as is already done in the L2 cache;
- invalidate speculative lines on a violation;
- reset speculative bits on epoch commit, and flush non-speculative lines from the victim cache;
- only speculative lines are placed in the speculative victim cache when evicted, non-speculative lines are evicted in the normal manner.

When an access hits in the L2 cache and the line has replicas, the victim cache must also be probed to determine if it contains relevant replicas. If these victim cache probes happen too often then there are two optimizations which can be done to minimize their frequency: first, only probe the victim cache if no line which is modified by the current epoch e is located in the L2 cache; and second, have a *may have victim* flag associated with each cache set which indicates that a line was evicted to the victim cache from that set in the past. Since the victim cache only caches speculative lines, the victim cache is expected to empty periodically, when the program runs a non-TLS-parallelized portion of code. When this happens all of the *may have victim* flags can be flash invalidated.

In our experiments with database software we found that when using a 2MB 4-way set associative L2 cache a 25-entry speculative victim cache was sufficient to contain all overflowed state.

4.2 Using TLS to Incrementally Parallelize Database Transactions

As we discussed earlier in Chapter 3, the epochs we extract from TPC-C transactions are large (the average epoch has 7,574–489,877 dynamic instructions) and contains many data dependences (the average epoch in NEW ORDER performs 292 loads which depend on values generated by the immediately previous epoch). In Chapter 3 we saw an iterative process for removing performance critical data dependences: (i) execute the speculatively-parallelized transaction on a TLS system; (ii) use profile feedback to identify the most performance critical dependences; (iii) modify the DBMS code to avoid violations caused by those dependences; and (iv) repeat. This process allows the programmer to treat parallelization of transactions as a form of *performance tuning*: a profile guides the programmer to the performance hot spots, and extra speed is obtained by modifying only the critical regions of code. Going through this process reduces the total number of data dependences between epochs (from 292 dependent loads per epoch to 75 dependent loads for NEW ORDER), but more importantly removes dependences from critical path.

The hardware assists in this iterative performance tuning process in two ways: (i) it provides mechanisms to track most performance-critical dependences; and (ii) it provides a mechanism for *escaping* speculation to avoid violations caused by database operations which are safe to execute in parallel (described further in Section 2.5).

4.2.1 Profiling Violated Inter-Epoch Dependences

To track which load and store program counter (PC) pairs cause the most harmful dependence violations we could use a simulator or a software instrumentation pass. However, hardware support for such profiling would be preferable and would only require a few extensions to basic TLS hardware support, as described by the following. Each processor must maintain an *exposed load table* [61]—a moderate-sized direct-mapped table of PCs, indexed by cache tag, which is updated with the PC of every speculative load which is *exposed* (i.e., has not been preceded in the current sub-epoch by a store to the same location—as already tracked by the basic TLS support). Each processor also maintains cycle counters which measure the duration of each sub-epoch.

When the L2 dependence tracking mechanism observes that a store has caused a violation: (i) the store PC is requested from the processor that issued the store; (ii) the corresponding load PC is requested from the processor that loaded the cache line (this is already tracked by the TLS mechanism), and the cache line tag is sent along with the request. That processor uses the tag to look-up the PC of the corresponding exposed load, and sends the PC along with the sub-epoch cycles back to the L2; in this case the cycle

```
①  if(some_work()) {
②     escape_speculation();
③     p = malloc(50);
④     on_violation_call(free, p);
⑤     resume_speculation();
    }
⑥  some_more_work();
```

Figure 4.4: Wrapper for the `pin_page` function which allows the ordering between epochs to be relaxed.

count represents failed speculation cycles. At the L2, we maintain a list of load/store PC pairs, and the total failed speculation cycles attributed to each. When the list overflows, we want to reclaim the entry with the least total cycles. Finally, we require a software interface to the list, in order to provide the programmer with a profile of problem load/store pairs, who can use the cycle counts to order them by importance.

4.2.2 Hardware Support for Escaping Speculation

It is easiest to explain escaping speculation with an example. Consider the code in Figure 4.4. In line ①, `some_work` runs speculatively. In line ② speculation is escaped, so the call to `malloc` in line ③ runs non-speculatively. Since `malloc` runs non-speculatively, any loads it performs will not cause violations. In line ④ we register a *recovery function* with the hardware. The hardware maintains a short list of recovery functions and parameters, and if a violation occurs the hardware invokes all of the recovery functions on the list. In this example, if speculation fails we call `free` to release the memory. Line ⑤ then resumes speculation so that `some_more_work` is run speculatively.

To escape speculation, the hardware temporarily treats the executing epoch as *non-speculative.* This means that all loads by the escaped epoch return committed memory state, and not speculative memory state. All stores performed by the epoch are not buffered by the TLS mechanism, and are immediately visible to all other epochs. A side effect of this is if an escaped epoch writes to a location speculatively loaded by any speculative epoch, *including itself*, it can cause that epoch to be violated. The only communication between the speculative execution preceding the escaped speculation and the escaped epoch is through registers, which can be carefully checked for invalid values caused by misspeculation. To avoid false dependences caused by writing temporary variables and return addresses to the stack, the escaped epoch should use a separate stack (using a mechanism such as *stacklets* [13, 59]).[3] If an escaped epoch is violated, it does not restart until speculation resumes—this way the escaped code does not have to be written to handle unexpected interrupts due to violations.

When an escaped epoch loads data into the L1 cache it may perform loads which evict lines which were speculatively modified by the current epoch. If the epoch resumes speculative execution and then loads the same line it must get the evicted copy of the line, and not the clean copy which just replaced it. To avoid this situation, we add one more state bit to each cache line in the L1 cache, called the *stale* bit. When a clean replica is retrieved from the L2 cache (and a speculative version exists) then the L2 cache indicates that the line is stale in its response, and the stale bit gets set for this line in the L1 cache. The next speculative read of this line will then miss to the L2 cache to retrieve the proper speculative version.

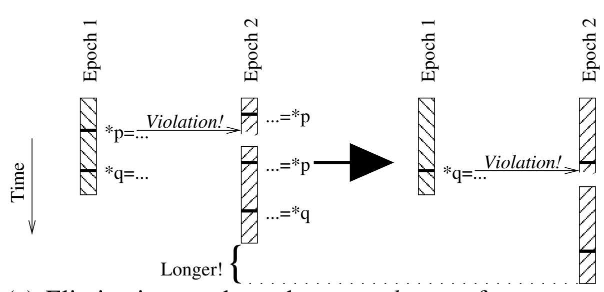
We found that the use of the *stale* bit caused speculative and clean replicas of cache lines to ping-pong to the L2 cache, dramatically increasing the number of L1 cache misses. This harmed performance, so to avoid this problem we modified the L1 cache to allow a limited form of replication—a set can hold both the speculative and clean replica version of a cache line (an alternative solution which should have similar performance is to put a victim cache underneath the L1 cache to catch these ping-ponging lines).

Since the escaped code takes non-speculative actions, the wrapper around it has to be carefully written to avoid causing harm to the rest of the program when mis-speculation happens. For example, a mis-speculating epoch may go into an infinite loop allocating memory. The mis-speculating epoch must not be allowed to allocate so much memory that allocation fails in the homefree epoch. This potential problem can be avoided through software: one way is to place limits on the amount of memory allocated by a speculative epoch. Another solution is to have homefree epochs first violate any speculative epochs (and hence have them release all resources) before allowing any allocation request to fail.

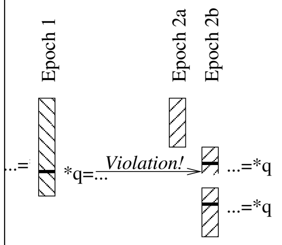
4.3 Tolerating Dependences with Sub-Epochs

When speculation fails for a large speculative epoch, the amount of work discarded is itself large, making this a costly event to be avoided. To tune performance, we want to allow the

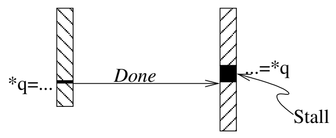
[3]TLS already assumes that each executing epoch has a private *stacklet* for storing local variables and the return addresses of called functions. If two epochs shared a common stack then an action as simple as calling a function would cause a violation, since each epoch would spill registers, write return addresses and write return values onto the stack. We assume that local stack writes are not intended to be shared between epochs (the compiler can check this assumption at compile time), and give each epoch a separate stacklet to use while it executes. To avoid conflicts between escaped execution and speculative execution, we allocate another stacklet for use by a epoch when it escapes speculation.



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



(b) Execution with sub-epochs.



(c) Idealized parallel execution.

Figure 4.5: Sub-epochs improve performance when dependences exist.

programmer to eliminate dependences one-by-one; but eliminating one dependence may expose an even-later dependence, which can potentially make performance *worse* (Figure 4.5(a)). Previous work has proposed data dependence predictors and value predictors to tolerate inter-epoch dependences [40, 61]: if a dependence between two epochs can be predicted, then it is automatically synchronized to avoid a violation; if the value used by a dependence is predictable, then a value predictor can avoid the data dependence altogether. In our database benchmarks the epochs are so large that they contain many dependences (between 20 and 75 dependent loads per epoch *after* optimization), hence any predictor would have to be very accurate to avoid violations.

In this thesis we do not rely on predictors to avoid violations—instead, we reduce the cost of violations by using *sub-epochs* (first introduced in Section 2.2). A sub-epoch is like a checkpoint during the execution of a epoch: when a violation occurs, execution rewinds back to the start of the sub-epoch which loaded the incorrect value (Figure 4.5(b)). When a violation occurs, TLS rewinds both the mis-speculated execution following the errant load and also rewinds the correct execution preceding the errant load—hence sub-epochs reduce the amount of correct execution which is rewound. With enough sub-epochs per epoch, TLS execution approximates an idealized parallel execution (Figure 4.5(c)) where data dependences limit performance by effectively stalling loads until the correct value is produced. Note that sub-epochs are *complimentary* to prediction: the use of sub-epochs reduces the penalty of a mis-prediction, and thus makes predictor design easier.

When we first experimented with sub-epochs we found that they also had a secondary effect: by reducing the penalty of a violation, sub-epochs make it easier for later epochs to fall into a state of "self-synchronization": if the start of an epoch is delayed by just enough, then backwards data dependences will be turned into forwards dependences. Forwards dependences do not cause violations, since aggressive update propagation commu-

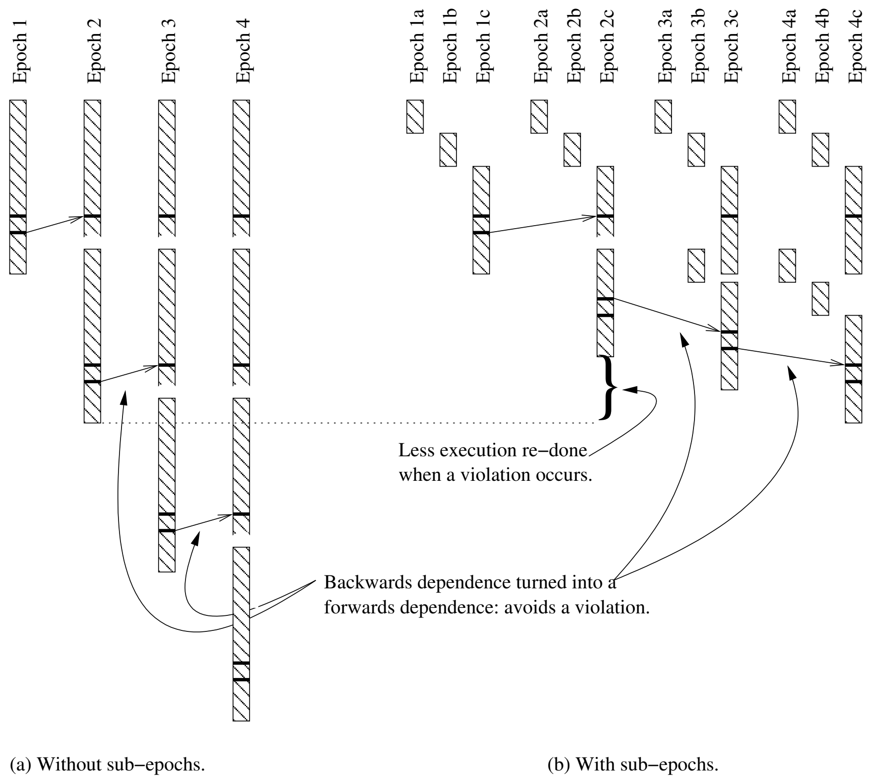
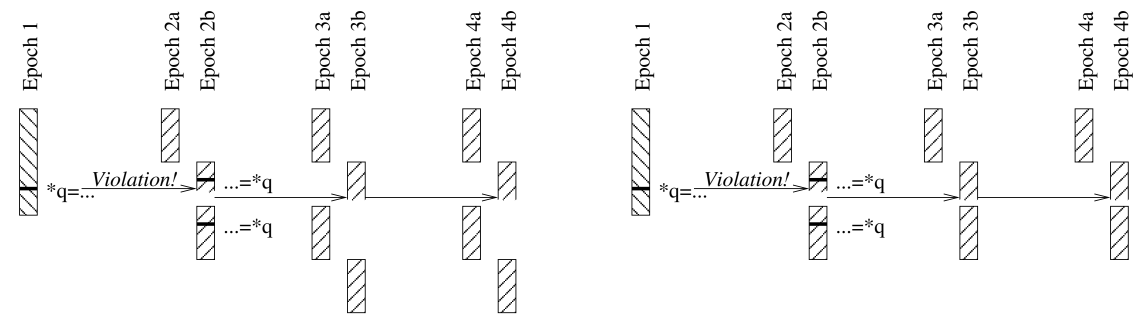


Figure 4.6: The two sources of performance improvement from sub-epochs.

nicates values between epochs. In Figure 4.6, we see that the addition of two sub-epochs allows the violated epochs to restart at slightly different points in their execution, which introduces just enough skew between epochs to avoid further violations.

4.3.1 Hardware Support for Sub-Epochs

Sub-epochs are implemented by allocating multiple hardware epoch contexts in the L2 cache for the execution of a single epoch. For example, assume that the L2 cache supports the execution of four epochs (and hence has four epoch contexts): to support two sub-epochs per epoch, we modify the L2 cache to support eight epoch contexts, and use the first two contexts for the first epoch, the next two for the second epoch, and so on. When each sub-epoch starts, a copy of the registers is made and all memory accesses from that point onwards use the next epoch context. Once all of the epoch contexts associated with a epoch are in use, no more sub-epochs can be created.



(a) All later sub-epochs restart. (b) Only dependent sub-epochs restart.

Figure 4.7: The effect of chain violations with and without sub-epoch dependence tracking.

In TLS, violations are detected between epoch contexts, so if each epoch context tracks a sub-epoch then a violation will specify which epoch *and sub-epoch* needs to be restarted. Since a given epoch's sub-epochs execute in-order, there will be no dependence violations between them. It is also unnecessary (for correctness) to make the L1 cache aware of the sub-epochs: when a violation occurs all speculative state in the L1 cache is invalidated, and the L1 cache retrieves any needed state from the L2 cache when the epoch re-executes. If invalidating *all* speculative state in the L1 cache on a violation causes too many cache misses then the L1 could be extended to track sub-epochs as well (in our experiments we have not found this to be a significant performance problem). Therefore no additional hardware is required to detect dependence violations between epochs at a sub-epoch granularity, other than providing the additional epoch contexts.

In our baseline TLS system, when a epoch is violated due to a data dependence we call this a *primary violation*; since later epochs may have consumed incorrect values generated by the primary violated epoch all later epochs are restarted with a *chain violation*. With sub-epochs this behavior is suboptimal, as illustrated in Figure 4.7(a). In this figure sub-epochs 3a and 4a are restarted, even though these epochs completed before sub-epoch 2b started. Since sub-epochs 3a and 4a could *not* have consumed any data from the restarted sub-epoch 2b, it is unnecessary to restart sub-epochs 3a and 4a. If the hardware tracked the temporal relationship between sub-epochs, then better performance would result, as shown in Figure 4.7(b). This temporal relationship between sub-epochs can be tracked with a *sub-epoch start table*, which is described in detail in the next section.

The sub-epochs within an epoch are executed in sequential order, so it is not necessary to check for violations between the epoch contexts used for a single epoch, and hence those checks can be omitted. It is also not necessary to make the L1 cache aware of the sub-epochs for correct execution: when a violation occurs all speculative state in the L1 cache is invalidated, and the L1 cache retrieves any needed state from the L2 cache when the epoch re-executes. If invalidating *all* speculative state in the L1 cache on a violation causes too many cache misses then the L1 could be extended to track sub-epochs as well (we have not found this to be a significant problem).

One IRB, ORB and violation recovery function list exists for each epoch, and they are appended to as an epoch executes. When a new sub-epoch is started the current IRB, ORB and violation recovery function list pointers are checkpointed (just like the registers), and they are restored on a violation.

Sub-epoch Start Table

To ensure that chain violations do not cause too much work to be redone we track the relationships between sub-epochs using the *sub-epoch start table*. The table records the sub-epochs which were executing for all later epochs when each sub-epoch begins. If the sub-epoch currently being executed by an epoch e is represented by S_e, then this table takes the following form:

$$T(e,p,s) = \text{The value of } S_e \text{ when epoch } p \text{ started sub-epoch } s$$

With this table, if epoch e receives a chain violation which indicates that epoch p just rewound to the start of sub-epoch s, then epoch e only needs to rewind to the start of sub-epoch $T(e,p,s)$.

This table must be maintained as sub-epochs start, new epochs begin, and violations occur. Each time a sub-epoch starts it records the sub-epoch being executed by all later epochs in the table.[4] If the epoch and sub-epoch starting are p_s and s_s then:

$$\forall e : T(e,p_s,s_s) := \begin{cases} S_e & \text{if } e \text{ is later then } p_s \\ T(e,p_s,s_s) & \text{otherwise} \end{cases}$$

When a new epoch starts executing it clears all entries which refer to it in the table. If the new epoch is e_n then:

$$\forall p,s : T(e_n,p,s) := 0$$

When a violation occurs, we reset the sub-epoch S_e to an earlier value, and must update table entries which point at sub-epochs which we have undone. If the violated epoch is e_v, and the updated sub-epoch is S_{e_new} then:

$$\forall p,s : T(e_v,p,s) := \min(S_{e\_new}, T(e_v,p,s))$$

[4]This can be done somewhat lazily, as long as the update is complete before any writes performed by a sub-epoch are made visible to any later epochs.

4.3.2 Choosing Sub-epoch Boundaries

As a given epoch executes, when should the system start each new sub-epoch? How many sub-epochs are necessary for good performance? The answer to these questions balances two competing factors: first, using more sub-epochs decreases the penalty of a violation. Second, adding hardware support for more sub-epochs is inexpensive, but not free—in the design outlined here each sub-epoch context requires two additional bits of storage per cache line in the L2 cache. By intelligently choosing sub-epoch boundaries we reduce the need for using many sub-epochs.

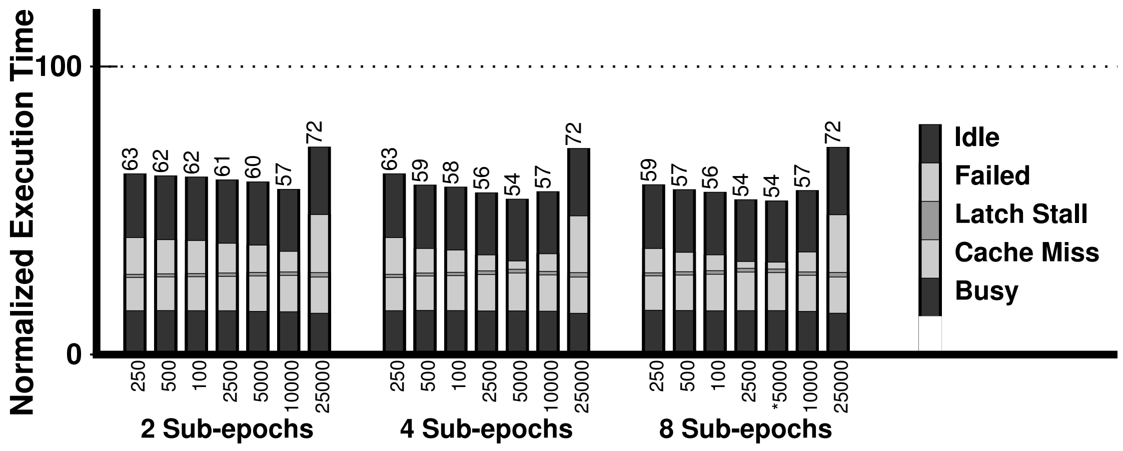
Inter-epoch dependences (and hence violations) are rooted at loads from memory, and so we want to start new sub-epochs just before loads. For each load, this leads to two important questions. First, is this load likely to cause a dependence violation? We want to start sub-epochs before loads which frequently cause violations, to minimize the amount of correct execution rewound in the common case. Previously proposed predictors can be used to detect such loads loads [61]. Second, if the load does cause a violation, how much correct execution will the sub-epoch avoid rewinding? If the load is near the start of the epoch or a previous sub-epoch, then a violation incurred at this point will have a minimal impact on performance. Instead, we would rather save the valuable sub-epoch context for a more troublesome load. A simple strategy that works well in practice is to start a new sub-epoch every nth speculative instruction—however, the key is to choose n carefully. We investigate several possibilities later in Section 4.4.1.

4.4 Experimental Results

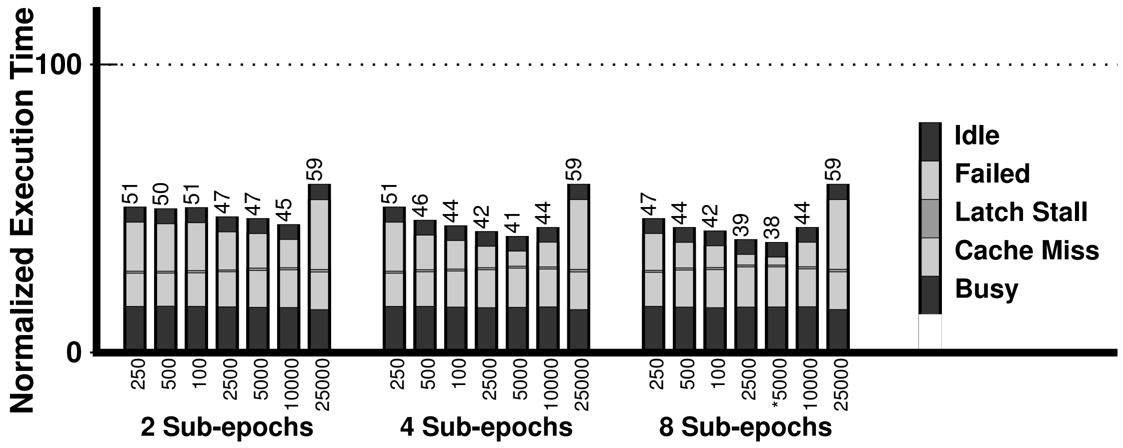
In this section we assume that the database system programmer has created a TLS-parallel version of TPC-C, and use it to evaluate the hardware design decisions presented in this thesis. For further information on how the programmer parallelizes the transactions, see Chapter 3. All of the results in this chapter use the same benchmarks and simulation infrastructure, which was described in detail in Section 3.4.

4.4.1 Sub-epoch Support

Adding sub-epochs to the hardware adds some complexity, but here we show that the extra cost is worth it. In the NO SUB-EPOCH bar in Figure 4.9 we disable support for sub-epochs. Compared to the performance of the BASELINE bar, sub-epoch support is clearly beneficial as it dramatically reduces the penalty of mis-speculation.

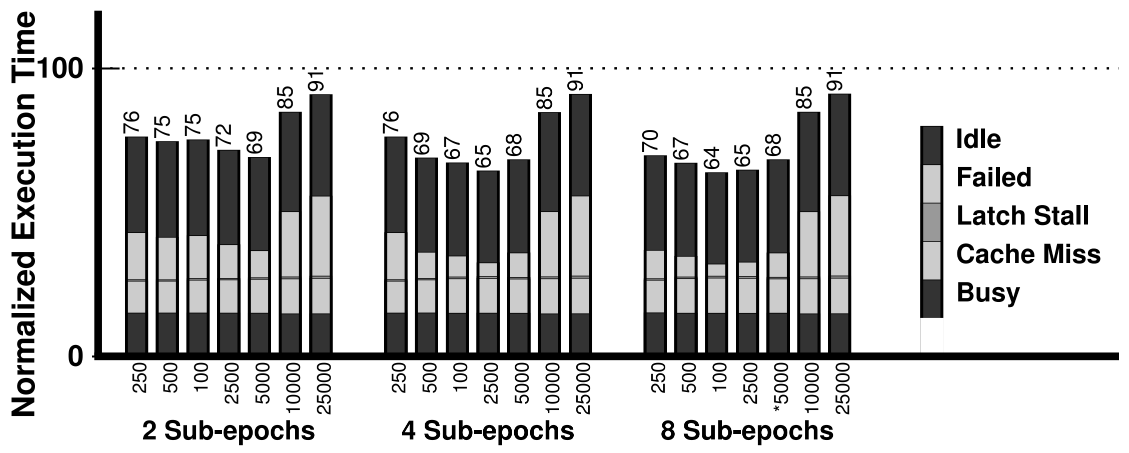


(a) NEW ORDER

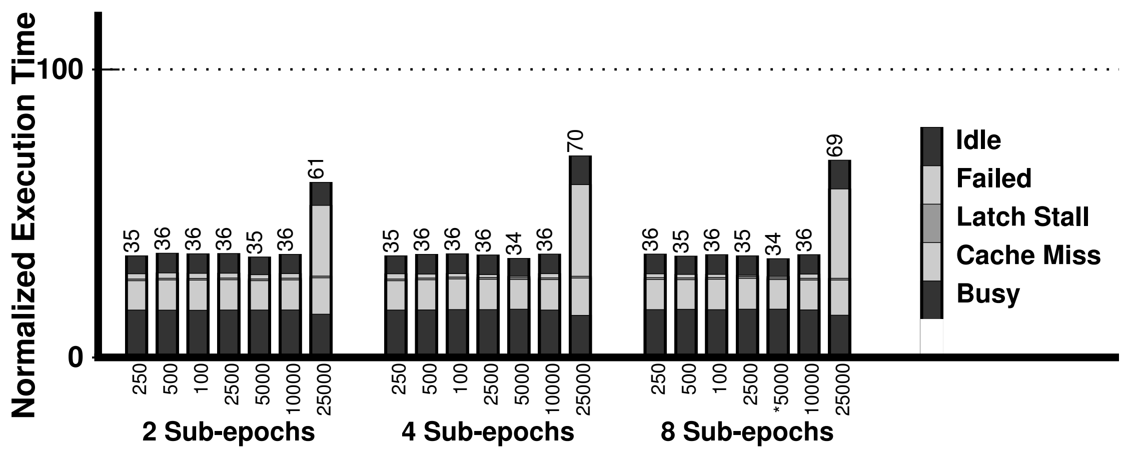


(b) NEW ORDER 150

Figure 4.8: Performance of optimized benchmark when varying the number of supported sub-epochs per epoch from 2 to 8, varying the number of speculative instructions per sub-epoch from 250 to 25000. The BASELINE experiment has 8 sub-epochs and 5000 speculative instructions per epoch.

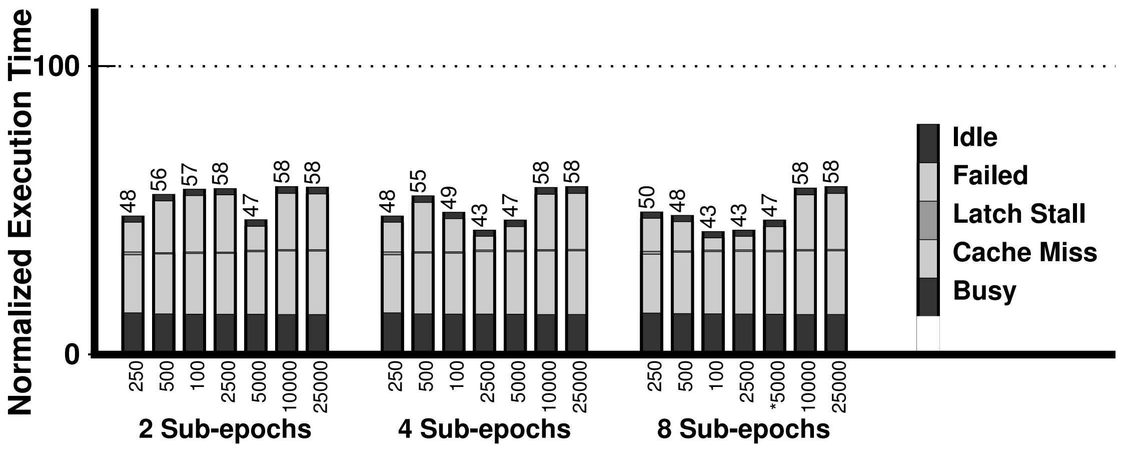


(c) DELIVERY



(d) DELIVERY OUTER

Figure 4.8: *Continued.*



(e) STOCK LEVEL

Figure 4.8: *Continued.*

Sub-epoch Size and Placement

How many sub-epochs are needed, and when should the system use them? In Figure 4.8 we show an experiment where we varied the number of sub-epochs available to the hardware, and varied the spacing between sub-epoch start points. We would expect that the best performance would be obtained if the use of sub-epochs is conservative, since this minimizes the number of replicate versions of each speculative cache line, and hence minimizes cache pressure. Since each sub-epoch requires a hardware epoch context, using a small number of sub-epochs also reduces the amount of required hardware. A sub-epoch would ideally start just before the first mis-speculating instruction in a epoch, so that when a violation occurs the machine rewinds no further than required.

If the hardware could predict the first dependence very accurately, then supporting 2 sub-epochs per epoch would be sufficient. With 2 sub-epochs the first sub-epoch would start at the beginning of the epoch, and the second one would start immediately before the load instruction of the predicted dependence. In our experiments we do not have such a predictor, and so instead we start sub-epochs periodically as the epoch executes.

In Figure 4.8 we vary both the number and size of the sub-epochs used for executing each transaction. Surprisingly, adding more sub-epochs does not seem to have a negative effect due to increased cache pressure. Instead, the additional sub-epochs serve to either increase the fraction of the epoch which is covered by sub-epochs (and hence protected from a large penalty if a violation occurs), or increase the density of sub-epoch start points within the epoch (decreasing the penalty of a violation).

We found that chosing a fixed distance between sub-epochs of 5000 dynamic instructions works quite well. This distance covers most of the NEW ORDER transaction with 8 sub-epochs per epoch. Closer inspection of both the epoch sizes listed in Table 3.2 and the graphs of Figure 4.8 reveals that instead of choosing a fixed sub-epoch size, a better strategy for choosing the sub-epoch size for small epochs is to measure the average epoch size and divide by the number of available sub-epochs.

One interesting case is DELIVERY OUTER, in Figure 4.8(d). In this case a data dependence early in the epoch's execution causes all but the non-speculative epoch to restart. With small sub-epochs the restart modifies the timing of the epoch's execution such that a data dependence much later in the epoch's execution occurs in-order, avoiding violations. Without sub-epochs, or with very large sub-epochs (such as the 25000 case in Figure 4.8(d)) this secondary benefit of sub-epochs does not occur.

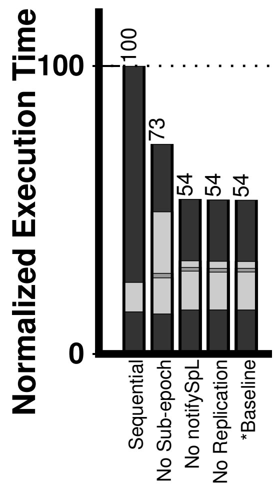
4.4.2 Cache Configuration

In this chapter we have introduced a new cache design for executing large, dependent epochs. In the next section we examine performance as optional features of the design are removed, and as design parameters are varied.

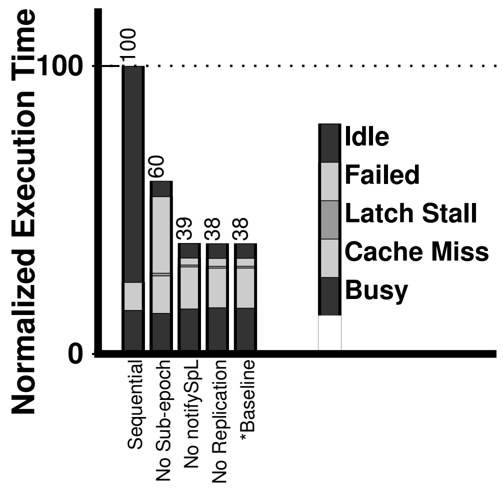
L1 Cache

When escaping speculation we found that the L1 cache often suffered from thrashing, caused by a line which was repeatedly loaded by speculative code, and then loaded by escaped code. To combat this effect we added replication to the L1 cache, which lets an L1 cache hold both a speculative and non-speculative version of a cache line simultaneously. In the NO REPL bar in Figure 4.9 we have removed this feature. If you compare it to the baseline, you can see that once the benchmark has been optimized this feature is no longer performance critical.

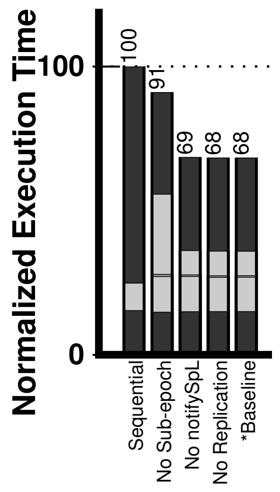
In our baseline design when the L1 cache receives a request to speculatively load a line, and a non-speculative version of the line already exists in the cache then the line is promoted to become speculative, and the L2 cache is notified through a *notify speculatively loaded* (*notifySpL*) message. This design optimistically assumes that the non-speculative line in the L1 cache has not been made obsolete by a more speculative line in the L2. If our optimism is misplaced then a violation will result. To see if this was the correct trade-off, in the NO NOTIFYSPL bar in Figure 4.9 we remove this message, and cause a



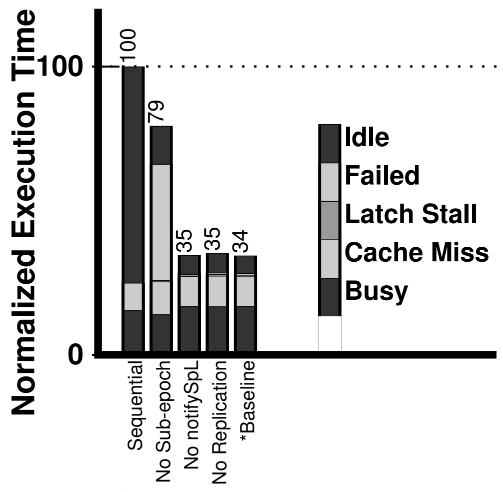
(a) NEW ORDER



(b) NEW ORDER 150

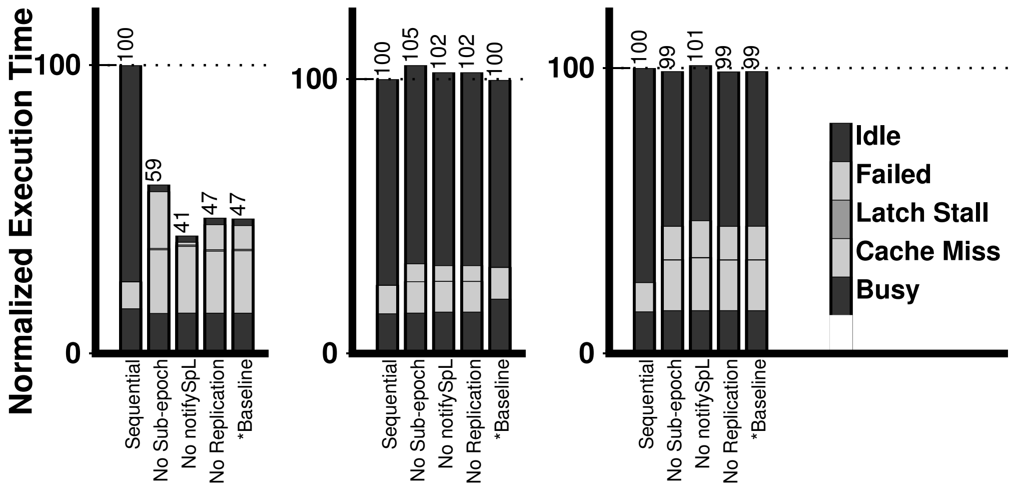


(c) DELIVERY



(d) DELIVERY OUTER

Figure 4.9: Performance of optimized benchmark without sub-epoch support, without NSL message, and without replication in the L1 cache.



(e) STOCK LEVEL (f) PAYMENT (g) ORDER STATUS

Bar	Explanation
Sequential	No modifications or TLS code added.
TLS-Seq	Optimized for TLS, but run on a single CPU.
No Sub-epoch	Baseline execution with sub-epoch support disabled.
No notifySpL	Baseline execution without the use of *notifySpL* messages from the L1 to L2 cache.
No Repl	Baseline execution without support for replication in the L1 cache.
Baseline	Execution on hardware described in this thesis.
No Spec	Upper bound—modified hardware to treat all speculative writes as non-speculative.

Figure 4.9: *Continued.*

Table 4.5: Maximum number of victim cache entries (lines) used versus L2 cache associativity.

Benchmark	4-way	8-way	16-way
NEW ORDER	54	4	0
NEW ORDER 150	64	39	0
DELIVERY	14	0	0
DELIVERY OUTER	62	4	0
STOCK LEVEL	40	0	0

speculative load of a line which exists non-speculatively in the L1 cache to be treated as an L1 cache miss. We see that the *notifySpL* message offers a very minor performance gain to NEW ORDER, NEW ORDER 150, DELIVERY and DELIVERY OUTER, but the optimism causes non-trivial slowdown for STOCK LEVEL. As a result, we conclude that unless the optimism that the *notifySpL* message offers is tempered, using the *notifySpL* message is a bad idea.

To quantify the impact of using a write-through (as opposed to a write-back) L1 cache, we ran the SEQUENTIAL benchmarks with two system configurations: on a system with write-back caches, and on our baseline system with a write-through cache. (All other system parameters are identical.) We found that with a write-back cache the write miss rate was roughly 1%—this means that 99 out of a hundred writes were not visible to the L2 cache. With a write-through design that filter is removed, and we observed that in the write-through model the L2 cache receives between 80 and 250 times as many update messages as it receives write-back messages in a write-back design. Since we configured our CPUs with ample write buffers and L1–L2 bandwidth, this does not significantly impact the performance of the machine. Note that providing this additional bandwidth on chip is practical: for example, the Intel Pentium 4 uses a 4-way set associative 8KB write-through L1 cache [21].

L2 Cache

We have added a speculative victim to our design to avoid violations caused by evicting speculative cache lines from the L2 cache. How large does it have to be? We expect that sub-epoch induced replication may exacerbate the problem of eviction induced violations.

In this study we used a 64-entry victim cache, but also measured how many entries are actually used in the cache. With our baseline 4-way L2 cache and using a 4-CPU machine, we see in Table 4.5 that only one experiment (NEW ORDER 150) uses all of the entries in the victim cache. If we increase the associativity of the L2 cache to 8-way then only 4 entries are ever used, and with a 16-way L2 the victim cache is never utilized. From this we conclude that the victim cache can be made quite small and remain effective. In addition, our experiments have found that increasing the L2 associativity (under the optimistic assumption that changing associativity has no impact on cycle time) has a less than 1% impact on performance.

4.5 Chapter Summary

- When applying TLS to database transactions the resulting epochs are large (7k–500k dynamic instructions per epoch), and contain many dependences between them (before optimization NEW ORDER averaged 292 exposed loads per epoch).
- To perform well, the hardware must be able to buffer these large epochs, and also be able to tolerate dependences between them without resorting to sequential execution.
- A design which stores speculative state in a shared L2 cache with a speculative victim cache was presented.
- A 2MB L2 cache with a 64-entry victim cache is sufficient to buffer even the largest epochs encountered.
- Storing speculative state in a shared cache allows dependent values to propagate to consumer epochs quickly, avoiding violations caused by slow value propagation.
- Hardware support for profiling TLS execution is presented. This hardware support enables the optimizations presented in previous chapters.
- Sub-epochs are implemented by adding additional epoch contexts to the hardware design, and adding a *sub-epoch start table* to track the temporal relationships between sub-epochs. We demonstrate that extending our TLS design to support sub-epochs is straightforward.
- We find that a straightforward scheme for choosing sub-epoch boundaries (periodically starting a new sub-epoch) is effective at eliminating the majority of the cycles wasted on failed speculation for the optimized benchmarks.
- Violations caused by overflow of the associative sets in the L2 cache can be avoided with a small victim cache. For a 4-way associative 2MB L2 cache 64 entries is sufficient, while for an 8-way cache 39 entries is sufficient to avoid violations. For a 16-way associative cache the victim cache is not required.

Chapter 5

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. In this thesis we have shown that 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.

We have focused on one such opportunity enabled by TLS: exploiting *intra-transaction* parallelism. Our experimental results demonstrate that we can speed up the *latency* of three of the five transactions in TPC-C by 46–66% by exploiting TLS on a chip multiprocessor with four CPU cores, or by 29–44% with two cores. This provides the database's scheduler with a new capability: with TLS the scheduler can use idle CPU cores to improve transaction latency. In contrast with previous approaches to exploiting intra-transaction 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 would generalize to other DBMSs. The hardware changes that enable this intra-transaction parallelism are feasible: we have shown a novel two-level cache protocol that: (i) allows a CMP with TLS support to buffer large epochs, (ii) allows the hardware to tolerate inter-epoch data dependences, and (iii) can profile those data dependences so the programmer can improve performance further. In short, we have shown that localized changes to transactions, the DBMS, and hardware work in concert to enable this new form of parallelism.

Previous work has shown that TLS works well for integer and numeric codes [18, 47, 55, 60], but the results in this thesis show that TLS works particularly well for database transactions. Why does TLS work so well for transactions? First, TLS enables an optimization process: performance improves dramatically when the programmer removes performance limiting data dependences. Previous work studying TLS has focused on programs that require fully automated parallelization techniques. In the domain of database systems it is worthwhile to apply hand-analysis to the DBMS, since optimizations to the DBMS code benefit *all* transactions that use the DBMS. Second, transactions spend a significant portion of their execution time executing routines with clearly defined interfaces, which can be modified to avoid data dependences (by escaping speculation, delaying their execution until homefree, or by substituting parallel routines). Third, the epochs found in transactions are quite large. In comparison to these large epochs the overheads of epoch initialization and committing are small.

Hardware that supports the concurrent execution of many threads is already here, and the number of concurrent threads will only grow with time. To harness this power requires parallel programs: writing parallel programs is hard, and rewriting existing programs to add parallelism may be even harder. Here we have shown a hybrid alternative: a program running on a TLS machine can alternate between sequential execution, parallel execution, and speculatively parallel execution as desired. We have demonstrated that even within a single loop the epochs can transition between speculative and non-speculative (escaped) execution to maximize performance. This flexibility makes it easier to create parallel programs, and to add parallelism to existing software.

The results in this thesis inspire many more interesting research questions: can our iterative performance tuning methodology be applied to other classes of programs? Does designing a TLS system which is *tolerant* of violations make it easier to design a compiler which generates TLS-parallel programs? Can TLS be applied elsewhere in the DBMS to improve performance? Can a single hardware design be created that efficiently supports both TLS-style execution within a transaction and hardware-supported optimistic concurrency control [20, 23] between transactions? In this thesis, we used simulation to gather a dependence profile for our optimization process—can hardware be designed that profiles dependences continuously without impacting performance? Can compilers be made intelligent enough to completely automate this optimization process?

TLS can provide exciting performance benefits to important domains beyond general-purpose and scientific computing. This thesis shows that it is possible to implement a complete system that can provide significant latency improvements for database transactions. These compelling results show that TLS support should be implemented in hardware, and database systems should use TLS to improve transaction latency.

Appendix A

Transaction Source Code

The results in this paper are derived from an implementation of a TPC-C like benchmark on top of the BerkeleyDB storage manager. The source code (before parallelization) for the 5 transactions used is listed for your reference—for complete benchmark source code in electronic format, please contact the authors.

A.1 DELIVERY

```
void delivery(DbEnv *env,
              Tables *tables,
              int thread_id,
              int warehouse,
              int carrier_id)
{
 for(int fail = 0;;) {
  try {
   AutoTxn txn(env);

   ///////////////////////////////////////////////////////
   // BEGIN TRANSACTION
```

For DELIVERY OUTER benchmark, TLS-parallelize this loop:

```
   for(int district = 0; district < DISTRICTS_PER_WAREHOUSE;
       district++) {

    int order = 0;
    int c_id = 0;
```

```
AutoCursor cur(tables->neworder.new_cursor(txn.tid()));
NewOrderTable::Key key(warehouse, district, 0);
Dbt key_dbt((void *)&key, sizeof(key));
Dbt row_dbt;

if(cur->get(&key_dbt, &row_dbt, DB_SET_RANGE | DB_RMW) == 0) {
 NewOrderTable::Key *key = (NewOrderTable::Key *)
     key_dbt.get_data();
 assert(key);

 assert(key->warehouse == warehouse);
 assert(key->district == district);

 order = key->order;

 // Delete the row from the NewOrder table:
 cur->del(0);

} else {
 // No orders to deliver!
 assert(0);

}

malloc_ptr<OrderTable::Row>
    order_r(tables->order.lookup(txn.tid(), true,
                                 warehouse, district, order));
assert(order_r);
c_id = order_r->c_id;
order_r->carrier_id = carrier_id;
tables->order.update(txn.tid(), warehouse, district, order,
                     order_r.get());

int total_amount = 0;
```

For DELIVERY benchmark, TLS-parallelize this loop:

```
for(int line_num = 0; line_num < order_r->ol_cnt; line_num++) {
 malloc_ptr<OrderlineTable::Row>
     orderline_r(tables->orderline.lookup(txn.tid(), true,
                                          warehouse,
                                              district,
                                              order, line_num));
      assert(orderline_r);
      strcpy(orderline_r->delivery_d, datetime);
      tables->orderline.update(txn.tid(), warehouse, district,
                               order, line_num,
                               orderline_r.get());
      total_amount += orderline_r->amount;

     }

     malloc_ptr<CustomerTable::Row>
         cust_r(tables->customer.lookup(txn.tid(), true, warehouse,
                                        district, c_id));
     assert(cust_r);
     cust_r->balance += total_amount;
     tables->customer.update(txn.tid(), warehouse, district,
                             c_id, cust_r.get());
    }

    // END TRANSACTION
    ////////////////////////////////////////////////////////////////

    // Done!  Commit transaction:
    txn.commit();

    break;
   }
   catch(DbException err) {
    env->err(err.get_errno(), "d: Caught exception, fail=d\n",
             thread_id, fail);

    if(fail++ > 4) {
     abort();
    }
   }
  }
 }
```

A.2 NEW ORDER

```
void new_order(DbEnv *env,
         Tables *tables,
         int thread_id,
         int warehouse,
         int district,
         int customer,
         int order_count,
         int item_w[],
         int item_id[],
         int item_qty[])
{
 for(int fail = 0;;) {
  try {
   AutoTxn txn(env);

   /////////////////////////////////////////////////////////////////
   // BEGIN TRANSACTION

   // Find customer and warehouse info:
   malloc_ptr<WarehouseTable::Row>
       ware_r(tables->warehouse.lookup(txn.tid(), false, warehouse));
   assert(ware_r);
   malloc_ptr<CustomerTable::Row>
       cust_r(tables->customer.lookup(txn.tid(), false, warehouse,
                                      district, customer));
   assert(cust_r);

   // Get and increment order number:
   malloc_ptr<DistrictTable::Row>
       dist_r(tables->district.lookup(txn.tid(), true, warehouse,
                                      district));
   assert(dist_r);

   int o_id = dist_r->next_o_id;
   dist_r->next_o_id++;

   tables->district.update(txn.tid(), warehouse,
                     district, dist_r.get());
```

```
// Create a new order:
OrderTable::Row orderRow;
orderRow.c_id = customer;
strcpy(orderRow.entry_d, datetime);
orderRow.carrier_id = 0;
orderRow.ol_cnt = order_count;
// Must adjust this code to support multiple warehouses
orderRow.all_local = 1;

tables->order.insert(txn.tid(), warehouse, district, o_id,
               &orderRow);

tables->neworder.insert(txn.tid(), warehouse, district, o_id);


// Process each item in the order:
```

For NEW ORDER benchmark, TLS-parallelize this loop:

```
for(int o_num=0; o_num < order_count; o_num++) {
 malloc_ptr<ItemTable::Row>
     item_r(tables->item.lookup(txn.tid(), false,
                           item_id[o_num]));

 if(!item_r) {
  throw InvalidItem();
 }

 malloc_ptr<StockTable::Row>
     stock_r(tables->stock.lookup(txn.tid(), true,
                             item_w[o_num],
                             item_id[o_num]));
 assert(stock_r);

 if(stock_r->quantity > item_qty[o_num]) {
  stock_r->quantity -= item_qty[o_num];
 } else {
  stock_r->quantity = stock_r->quantity -
      item_qty[o_num] + 91;
 }

 tables->stock.update(txn.tid(), item_w[o_num],
                     item_id[o_num], stock_r.get());
```

```
    float fprice = ((float)item_qty[o_num]) *
        ((float)item_r->price / 100.0) *
        ((float)(100 + ware_r->tax + dist_r->tax) / 100.0) *
        ((float)(100 - cust_r->discount) / 100.0);
    int price = (int)(fprice * 100.0);

    tables->orderline.insert(txn.tid(), warehouse, district,
                             o_id, o_num, item_id[o_num],
                             item_w[o_num], datetime,
                             item_qty[o_num], price,
                             "dist info");
   }

   // END TRANSACTION
   ////////////////////////////////////////////////////////////////

   // Done!  Commit transaction:
   txn.commit();
   break;
  }
  catch(DbException err) {
   env->err(err.get_errno(), "%d: Caught exception, fail=%d",
            thread_id, fail);

   if(fail++ > 4) {
    abort();
   }
  }
  catch(InvalidItem ii) {
   break;
  }
 }
}
```

A.3 Order Status

```
void order_status(DbEnv *env,
                  Tables *tables,
                  int thread_id,
                  int warehouse,
                  int district,
                  bool byname,
                  int customer,
                  char *cust_name,
                  OrderlineTable::Row *results[15])
{
 for(int fail = 0;;) {
  CustomerTable::Key *cust_k = NULL;
  CustomerTable::Row *cust_r = NULL;

  try {
   AutoTxn txn(env);

   /////////////////////////////////////////////////////////////////
   // BEGIN TRANSACTION

   {// New scope for autocursor destruction
    if(byname) {
     malloc_ptr<CustomerTable::Elem> matches(
         (CustomerTable::Elem *)
         malloc(sizeof(CustomerTable::Elem) * 3000));
     int num_matches = 0;

     AutoCursor cur(tables->custbyname.new_cursor(txn.tid()));
     CustByNameTable::Key key(warehouse, district, cust_name);
     Dbt key_dbt((void *)&key, sizeof(key));
     Dbt pkey_dbt;
     Dbt row_dbt;

     pkey_dbt.set_flags(DB_DBT_MALLOC);
     row_dbt.set_flags(DB_DBT_MALLOC);

     if(cur->pget(&key_dbt, &pkey_dbt, &row_dbt, DB_SET) == 0) {
      do {
       CustomerTable::Key *key = (CustomerTable::Key *)
      pkey_dbt.get_data();
  CustomerTable::Row *row = (CustomerTable::Row *)
      row_dbt.get_data();
  assert(key);
  assert(row);

  if(key->warehouse == warehouse &&
     key->district == district &&
     strcmp(row->last, cust_name) == 0) {
   assert(num_matches < 3000);
   matches[num_matches].key = key;
   matches[num_matches].row = row;
   num_matches++;
  } else {
   free(key);
   free(row);
   break;
  }
 } while(cur->pget(&key_dbt, &pkey_dbt, &row_dbt,
                  DB_NEXT) == 0);
}

assert(num_matches > 0);

qsort(matches.get(), num_matches,
      sizeof(CustomerTable::Elem),
      CustomerTable::compare_first);

int midpoint_match;
if(num_matches % 2) {
 midpoint_match = num_matches / 2;
} else {
 midpoint_match = (num_matches+1) / 2;
}

for(int n = 0; n < num_matches; n++) {
 if(n == midpoint_match) {
  cust_r = matches[n].row;
  cust_k = matches[n].key;
 } else {
  free(matches[n].row);
```

```
   free(matches[n].key);
  }
 }

 customer = cust_k->customer;
} else {
 cust_r =
     tables->customer.lookup(txn.tid(), true,
                             warehouse, district, customer);
 assert(cust_r);
}

// Find last order by this customer in the order table:
AutoCursor cur(tables->orderbycust.new_cursor(txn.tid()));
OrderByCustTable::Key key(warehouse, district, customer);
Dbt key_dbt((void *)&key, sizeof(key));
Dbt pkey_dbt;
Dbt row_dbt;
int order_id = -1;

pkey_dbt.set_flags(DB_DBT_MALLOC);
row_dbt.set_flags(DB_DBT_MALLOC);

if(cur->pget(&key_dbt, &pkey_dbt, &row_dbt, DB_SET) == 0) {
```

For ORDER STATUS benchmark, TLS-parallelize this loop:

```
 do {
  OrderTable::Key *key = (OrderTable::Key *)
      pkey_dbt.get_data();
  OrderTable::Row *row = (OrderTable::Row *)
      row_dbt.get_data();
  assert(key);
  assert(row);

  if(key->warehouse == warehouse &&
     key->district == district &&
     row->c_id == customer) {
   if(key->order > order_id) {
    order_id = key->order;
   }
  } else {
   free(key);
```

```
    free(row);
    break;
   }
   free(key);
   free(row);
 } while(cur->pget(&key_dbt, &pkey_dbt, &row_dbt,
                   DB_NEXT) == 0);
}

assert(order_id != -1);

AutoCursor ocur(tables->orderline.new_cursor(txn.tid()));
OrderlineTable::Key ol_key(warehouse, district, order_id, 0);
Dbt ol_key_dbt((void *)&ol_key, sizeof(ol_key));
Dbt ol_row_dbt;

ol_key_dbt.set_flags(DB_DBT_MALLOC);
ol_row_dbt.set_flags(DB_DBT_MALLOC);

int i = 0;

if(ocur->get(&ol_key_dbt, &ol_row_dbt, DB_SET_RANGE) == 0) {
```
For ORDER STATUS benchmark, TLS-parallelize this loop:
```
 do {
   OrderlineTable::Key *key = (OrderlineTable::Key *)
        ol_key_dbt.get_data();
   OrderlineTable::Row *row = (OrderlineTable::Row *)
        ol_row_dbt.get_data();
   assert(key);
   assert(row);

   if(order_id != key->order_id ||
      district != key->district ||
      warehouse != key->warehouse) {
    free(key);
    free(row);
    break;
   }

   assert(i < 15);
   results[i] = row;
```

```
      i++;

      free(key);
     } while(ocur->get(&ol_key_dbt, &ol_row_dbt, DB_NEXT) == 0);
    } else {
     assert(0);
    }
   }
   // END TRANSACTION
   ////////////////////////////////////////////////////////////////

   // Done!  Commit transaction:
   txn.commit();

   // Free all allocated memory:
   free(cust_r);
   free(cust_k);
   break;
  }
  catch(DbException err) {
   env->err(err.get_errno(), "%d: Caught exception, fail=%d\n",
           thread_id, fail);

   // Free all allocated memory:
   free(cust_r);
   free(cust_k);

   if(fail++ > 4) {
    abort();
   }
  }
 }
}
```

A.4 Payment

```
void payment(DbEnv *env,
             Tables *tables,
             int thread_id,
             int warehouse,
             int district,
             bool byname,
             int customer,
             char *cust_name,
             long amount)
{
 for(int fail = 0;;) {
  CustomerTable::Key *cust_k = NULL;
  CustomerTable::Row *cust_r = NULL;

  try {
   AutoTxn txn(env);

   ////////////////////////////////////////////////////////////////
   // BEGIN TRANSACTION

   malloc_ptr<WarehouseTable::Row>
       ware_r(tables->warehouse.lookup(txn.tid(), true, warehouse));
   assert(ware_r);
   ware_r->ytd += amount;
   tables->warehouse.update(txn.tid(), warehouse, ware_r.get());

   malloc_ptr<DistrictTable::Row>
       dist_r(tables->district.lookup(txn.tid(), true,
                                      warehouse, district));
   assert(dist_r);
   dist_r->ytd += amount;
   tables->district.update(txn.tid(), warehouse,
                           district, dist_r.get());

   if(byname) {
    malloc_ptr<CustomerTable::Elem> matches(
        (CustomerTable::Elem *)malloc(sizeof(CustomerTable::Elem) *
                                      3000));
    int num_matches = 0;
 AutoCursor cur(tables->custbyname.new_cursor(txn.tid()));
 CustByNameTable::Key key(warehouse, district, cust_name);
 Dbt key_dbt((void *)&key, sizeof(key));
 Dbt pkey_dbt;
 Dbt row_dbt;

 pkey_dbt.set_flags(DB_DBT_MALLOC);
 row_dbt.set_flags(DB_DBT_MALLOC);

 if(cur->pget(&key_dbt, &pkey_dbt, &row_dbt,
              DB_SET_RANGE | DB_RMW) == 0) {
  do {
   CustomerTable::Key *key = (CustomerTable::Key *)
       pkey_dbt.get_data();
   CustomerTable::Row *row = (CustomerTable::Row *)
       row_dbt.get_data();
   assert(key);
   assert(row);

   if(key->warehouse == warehouse &&
      key->district == district &&
      strcmp(row->last, cust_name) == 0) {
    assert(num_matches < 3000);
    matches[num_matches].key = key;
    matches[num_matches].row = row;
    num_matches++;
   } else {
    free(key);
    free(row);
    break;
   }
  } while(cur->pget(&key_dbt, &pkey_dbt, &row_dbt,
                   DB_NEXT | DB_RMW) == 0);
 }

assert(num_matches > 0);

qsort(matches.get(), num_matches, sizeof(CustomerTable::Elem),
      CustomerTable::compare_first);
```

```
int midpoint_match;
if(num_matches % 2) {
 midpoint_match = num_matches / 2;
} else {
 midpoint_match = (num_matches+1) / 2;
}

for(int n = 0; n < num_matches; n++) {
 if(n == midpoint_match) {
  cust_r = matches[n].row;
  cust_k = matches[n].key;
 } else {
  free(matches[n].row);
  free(matches[n].key);
 }
 }
} else {
 cust_r =
     tables->customer.lookup(txn.tid(), true,
                             warehouse, district, customer);
 assert(cust_r);
}
```

For PAYMENT benchmark, start one epoch here...

```
cust_r->balance += amount;

if(strstr(cust_r->credit, "BC")) {
 char new_data[500];
 sprintf(new_data, "| %4d %2d %4d %2d %4d $%7.2f %12s",
         byname ? cust_k->customer : customer,
         district, warehouse,
         district, warehouse, ((float)amount)/100.0, datetime);
 strncat(new_data, cust_r->data, 500 - strlen(new_data));
 strcpy(cust_r->data, new_data);
}

tables->customer.update(txn.tid(), warehouse, district, customer,
                        cust_r);
```

...and the next one here:

```
char h_data[25];
```

```
   strncpy(h_data, ware_r->name, 10);
   h_data[10] = '\0';
   strncat(h_data, dist_r->name, 10);
   h_data[20] = '\0';
   h_data[21] = '\0';
   h_data[22] = '\0';
   h_data[23] = '\0';

   tables->history.insert(txn.tid(), warehouse, district, customer,
                          district, warehouse, datetime, amount,
                          h_data);

   // END TRANSACTION
   ////////////////////////////////////////////////////////////////

   // Done!  Commit transaction:
   txn.commit();

   // Free all allocated memory:
   free(cust_r);
   free(cust_k);
   break;
  }
  catch(DbException err) {
   env->err(err.get_errno(), "%d: Caught exception, fail=%d\n",
            thread_id, fail);

   // Free all allocated memory:
   free(cust_r);
   free(cust_k);

   if(fail++ > 4) {
    abort();
   }
  }
 }
}
```

A.5 Stock Level

```
void stock_level(DbEnv *env,
                 Tables *tables,
                 int thread_id,
                 int warehouse,
                 int district,
                 int threshold,
                 int *low_stock_cnt)
{
 for(int fail = 0;;) {
  try {
   AutoTxn txn(env);

   ////////////////////////////////////////////////////////////////
   // BEGIN TRANSACTION

   int low_stock = 0;

   {// New scope for autocursor destruction
    malloc_ptr<DistrictTable::Row>
        dist_r(tables->district.lookup(txn.tid(), false, warehouse,
                                       district));
    assert(dist_r);
    int o_id = dist_r->next_o_id;

    AutoCursor cur(tables->orderline.new_cursor(txn.tid()));
    OrderlineTable::Key key(warehouse, district, o_id - 20, 0);
    Dbt key_dbt((void *)&key, sizeof(key));
    Dbt row_dbt;

    std::set<int, ltint> found_items;

    if(cur->get(&key_dbt, &row_dbt, DB_SET_RANGE) == 0) {
```

For Stock Level benchmark, TLS-parallelize this loop:

```
     do {
      OrderlineTable::Key *key = (OrderlineTable::Key *)
          key_dbt.get_data();
      OrderlineTable::Row *row = (OrderlineTable::Row *)
          row_dbt.get_data();
      assert(key);
```

```
    assert(row);

    if(key->warehouse != warehouse ||
       key->district != district ||
       key->order_id > o_id) {
     break;
    }
    assert(key->order_id >= o_id - 20);

    if(found_items.find(row->i_id) ==
       found_items.end()) {
     found_items.insert(row->i_id);

     malloc_ptr<StockTable::Row>
         stock_r(tables->stock.lookup(txn.tid(),
                                      false,
                                      warehouse,
                                      row->i_id));
     assert(stock_r);

     if(stock_r->quantity < threshold) {
      low_stock++;
     }
    }
   } while(cur->get(&key_dbt, &row_dbt, DB_NEXT) == 0);
  }
 }

 // END TRANSACTION
 ////////////////////////////////////////////////////////////////

 // Done!  Commit transaction:
 txn.commit();

 *low_stock_cnt = low_stock;

 break;
}
catch(DbException err) {
 env->err(err.get_errno(), "%d: Caught exception, fail=%d\n",
          thread_id, fail);
    if(fail++ > 4) {
     abort();
    }
   }
  }
 }
```

Appendix B

L2 Coherence Actions

This Appendix contains pseudo-code which describes in detail the algorithms used to select the appropriate cache line (amongst a set of replicas) and check for violations when the L2 cache receives a reference.

```
// What happens at a high level when the L1 cache receives a
// request from a L1 cache:
(Bool is_clean, Line line)
request_from_L1(Addr addr,
                Ref ref,
                EpochNum epoch)
{
  Tag tag = addr2tag(addr);
  CacheSet set = cache.set_lookup(addr2set(addr));
  LineSet lines = set.matches(tag) union victim_cache.matches(tag);

  (Bool is_hit, LinePtr linep) = select_line(lines, ref, epoch, set);

  if(!is_hit) {
    linep = set.lru();
    linep.evict();

    if(lines != empty_set && ref == UpSp) {
      (EpochNum most_recent_epoch, LinePtr most_recent_line) =
        find_most_recent(lines, epoch);

      if(most_recent_epoch != -1) {
        // In cache replicate:
```

```
        linep.copy(most_recent_line);
      }
    }

    linep.miss_to_next_cache();
  }

  // Note: does not show coherence messages sent to other L2 caches,
  // nor does this show invalidations sent to L1 caches:
  switch(ref) {
  case R:
    return (true, linep);
    break;

  case RSp:
    linep.set_SL(epoch);
    return ((linep.SM_bits() == empty_set), linep);
    break;

  case Up:
    linep.update();
    break;

  case UpSp:
    linep.update();
    linep.set_SM(epoch);
    break;

  case notifySL:
    linep.set_SL(epoch);
    break;
  }

  return (true, linep);
}

// How a line is selected from the set of replicas:
(Bool is_hit, LinePtr linep)
select_line(LineSet lines,
            Ref ref,
            EpochNum epoch,
            CacheSet set)
{
  if(lines != empty_set) {
    check_for_violations(lines, ref, epoch);

    (Bool is_compat, LinePtr linep) = compatible(lines, ref, epoch);

    if(is_compat) {
      if(victim_cache.contains(linep)) {
        LinePtr evicted_linep = set.lru();
        evicted_linep.evict();
        linep.move_to(evicted_linep);
      }
      return (true, linep);
    }
  }

  return (false, NULL);
}


// Given a reference and a set of matching lines, are there any
// violations?
void
check_for_violations(LineSet lines,
                     Ref ref,
                     EpochNum epoch)
{
  switch(ref) {
  case Up:
    foreach(line in lines) {
      if(line.SL_bits() != empty_set) {
        violate(bits2epochSet(line.SL_bits()));
      }
    }
    break;

  case UpSp:
    foreach(line in lines) {
      if(line.SL_bits((epoch+1)...infinity) != empty_set) {
        violate(bits2epochSet(line.SL_bits((epoch+1)...infinity)));
      }
    }
    break;

  case notifySL:
    foreach(line in lines) {
      if(line.SM_bits(0...(epoch-1)) != empty_set) {
        violate(epoch);
      }
    }
    break;

  case R:
  case RSp:
    break;
  }
}


// Given a reference and a set of matching lines, which line should we
// use?
(Bool is_compat, LinePtr linep)
compatible(LineSet lines,
           Ref ref,
           EpochNum epoch)
{
  switch(ref) {
  case R:
  case Up:
    foreach(line in lines) {
      if(line.SM_bits() == empty_set) {
        return (true, &line);
      }
    }
    break;

  case RSp:
  case notifySL:
    (EpochNum most_recent_epoch, LinePtr most_recent_line) =
      find_most_recent(lines, epoch);
    if(most_recent_epoch != -1) {
      return (true, &most_recent_line);
    }
    break;

  case UpSp:
    foreach(line in lines) {
      if(line.SM_bits(epoch...epoch) != empty_set) {
        return (true, &line);
      }
    }
    foreach(line in lines) {
      if(line.SM_bits() == empty_set &&
         line.SL_bits(0...(epoch-1)) == empty_set) {
        return (true, &line);
      }
    }
    break;
  }

  return (false, NULL);
}

// Find the line in the given set which has been modified by the most
// recent epoch which is earlier than or equal to the given epoch:
(EpochNum most_recent_epoch, LinePtr most_recent_line)
find_most_recent(LineSet lines,
                 EpochNum epoch)
{
  EpochNum most_recent_epoch = -1;
  LinePtr most_recent_line;

  foreach(line in lines) {
    if(line.SM_bits() == empty_set && most_recent == -1) {
      most_recent_line = &line;
      most_recent_epoch = 0;
    } else {
      if(line.SM_bits(0...epoch) != empty_set &&
         bit2epoch(line.SM_bits(0...epoch)) > most_recent_epoch) {
        most_recent_line = &line;
        most_recent_epoch = bit2epoch(line.SM_bits(0...epoch));
      }
    }
  }
  return (most_recent_epoch, most_recent_line);
}
```

Bibliography

[1] H. Akkary and M. Driscoll. A dynamic multithreading processor. In *MICRO-31*, December 1998. 10

[2] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2003. 11

[3] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the 11th HPCA*, 2005. 12

[4] Arvind and D. Culler. Dataflow architectures. In *Annual Review in Computer Science*, volume 1, pages 225–253, 1986. 17

[5] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *Proceedings of the 14th SPAA*, August 2002. 8

[6] M. Cintra, J. Martínez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *ISCA 27*, June 2000. 62

[7] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order commit processors. In *Proceedings of the 10th HPCA*, February 2004. 11

[8] E.D. Berger and K.S. McKinley and R.D. Blumofe and P.R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th ASPLOS*, 2000. 45

[9] M. Franklin and G. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5), May 1996. 10

[10] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259. ACM Press, 1987. ISBN 0-89791-236-5. doi: http://doi.acm.org/10.1145/38713.38742. 46

[11] M. Garzarán, M. Prvulovic, J. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. In *Proceedings of the 9th HPCA*, February 2003. 10, 16, 62

[12] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990. 27

[13] S. Goldstein, K. Schauser, and D. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, August 1996. 74

[14] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative versioning cache. In *Proceedings of the 4th HPCA*, February 1998. 10, 11, 62, 67

[15] J. Gray. *The Benchmark Handbook for Transaction Processing Systems*. Morgan-Kaufmann Publishers, Inc., 1993. 1, 29, 30

[16] M. Gupta and R. Nim. Techniques for speculative run-time parallelization of loops. In *Supercomputing '98*, November 1998. 10

[17] R. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, 1985. 8

[18] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro Magazine*, March-April 2000. 3, 8, 10, 62, 90

[19] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). In *Proceedings of the 11th ASPLOS*, October 2004. 8

[20] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31th ISCA*, June 2004. 8, 90

[21] B. Hayes. Differences in optimizing for the pentium 4 processor versus the pentium iii processor. `http://www.intel.com/cd/ids/developer/asmo-na/eng/44010.htm`, 2005. 86

[22] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach: Second Edition*. Morgan Kaufmann, 1996. 40

[23] M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th ISCA*, 1993. 10, 90

[24] IBM Corporation. *IBM DB2 Universal Database Administration Guide: Performance*. IBM Corporation, 2004. 1

[25] T. Johnson, R. Eigenmann, and T. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proc. ACM SIGPLAN 04 Conference on Programming Language Design and Implementation*, June 2004. 8

[26] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th ISCA*, pages 364–373, May 1990. 68, 71

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

[28] T. Knight. An architecture for mostly functional languages. In *Proceedings of the ACM Lisp and Functional Programming Conference*, pages 500–519, August 1986. 8

[29] H. Kung and J. Robinson. On optimistic methods for concurrency control. *ACM TODS*, pages 213–226, June 1981. 10, 43

[30] A. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *ISCA 27*, June 2000. 66

[31] S. Mahlke, W. Chen, J. Gyllenhaal, and W. Hwu. Compiler code transformations for superscalar-based high-performance systems. In *In Proceedings of the International Conference on Supercomputing*, 1992. 33

[32] P. Marcuello and A. González. Clustered speculative multithreaded processors. In *Proc. of the ACM Int. Conf. on Supercomputing*, June 1999. 10

[33] J. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, November 2002. 11

[34] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993. 49

[35] D. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Priority mechanisms for OLTP and transactional web applications. In *Proceedings of the IEEE International Conference on Data Engineering*, March 2004. 7

[36] D. T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Improving preemptive prioritization via statistical characterization of OLTP locking. In *Proceedings of the IEEE International Conference on Data Engineering*, 2005. 7

[37] J. Miller and H. Lau. *Microsoft SQL Server 2000 Resource Kit*, chapter RDBMS Performance Tuning Guide for Data Warehousing, pages 575–653. Microsoft Press, 2001. 1

[38] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, March 1992. 27, 46

[39] G. Morrisett and M. Herlihy. Optimistic parallelization. Technical Report CMU-CS-93-171, School of Computer Science, Carnegie Mellon University, October 1993. 10

[40] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th ISCA*, June 1997. 11, 75

[41] J. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, MIT, April 1981. Tech Rep MIT/LCS/TR-260. Also available from MIT press, 1985. 20

[42] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the Summer Usenix Technical Conference*, June 1999. 29, 30

[43] K. Olukotun, L. Hammond, and M. Willey. Improving the performance of speculatively parallel applications on the hydra cmp. In *13th Annual ACM International Conference on Supercomputing*, 1999. 11

[44] C. L. Ooi, S. W. Kim, I. Park, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *In Proceedings of the International Conference on Supercomputing*, June 2001. 8

[45] J. Oplinger, D. Heine, and M. Lam. In search of speculative thread-level parallelism. In *Proceedings of PACT '99*, October 1999. 16

[46] M. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *The ACM SIGPLAN 2003 Symposium on Principles & Practice of Parallel Programming*, June 2003. 8, 16, 62

[47] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas. Removing architectural bottlenecks to the scalability of speculative parallelization. In *Proceedings of the 28th ISCA*, June 2001. 4, 10, 11, 62, 68, 90

[48] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32th ISCA*, 2005. 12

[49] L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed System*, 10(2):160–172, 1999. 10

[50] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, December 1997. 10

[51] P. Rundberg and P.Stenstrom. Low-cost thread-level data dependence speculation on multiprocessors. In *Fourth Workshop on Multithreaded Execution, Architecture and Compilation*, December 2000. 10

[52] M. Rys, M. Norrie, and H. Schek. Intra-transaction parallelism in the mapping of an object model to a relational multi-processor system. In *Proceedings of the 22nd VLDB*, 1996. 9

[53] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM TODS*, 20(3):325–363, 1995. URL `citeseer.ist.psu.edu/shasha95transaction.html`. 9

[54] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., 2002. 43

[55] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd ISCA*, June 1995. 10, 62, 90

[56] Standard Performance Evaluation Corporation. The SPEC Benchmark Suite. `http://www.specbench.org`. 61

[57] J. Steffan. *Hardware Support for Thread-Level Speculation*. PhD thesis, Carnegie Mellon University, School of Computer Science, April 2003. 67, 70

[58] J. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallellization. In *Proceedings of the 4th HPCA*, February 1998. 10, 68

[59] J. Steffan, C. Colohan, and T. Mowry. Architectural support for thread-level data speculation. Technical Report CMU-CS-97-188, School of Computer Science, Carnegie Mellon University, November 1997. 10, 74

[60] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. A scalable approach to thread-level speculation. In *ISCA 27*, June 2000. 3, 10, 62, 63, 90

[61] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. Improving value communication for thread-level speculation. In *Proceedings of the 8th HPCA*, February 2002. 10, 11, 16, 62, 72, 75, 79

[62] Transaction Processing Performance Council. TPC benchmark C standard specification revision 5.4. `http://www.tpc.org`, April 2005. 30

[63] M. Tremblay. MAJC: Microprocessor architecture for java computing. *HotChips '99*, August 1999. 3

[64] N. Tuck and D. M. Tullsen. Multithreaded value prediction. In *Proceedings of the 11th HPCA*, February 2005. 11

[65] T. Vijaykumar. *Compiling for the Multiscalar Architecture*. PhD thesis, University of Wisconsin-Madison, January 1998. 8, 16, 62

[66] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, April 1996. 49

[67] A. Zhai, C. Colohan, J. Steffan, and T. Mowry. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In *Proceedings of the 10th ASPLOS*, October 2002. 10

[68] A. Zhai, C. Colohan, J. Steffan, and T. Mowry. Compiler optimization of memory-resident value communication between speculative threads. In *International Symposium on Code Generation and Optimization*, March 2004. 10, 16

[69] C. Zuzarte. Personal communication, 2005. 1

Glossary

Aggressive update propagation: When an epoch stores to a memory location the new value is made available to later epochs as soon as possible. This avoids violations caused by forwards dependences., 14

Backwards dependence: A dependence between two epochs where the second epoch loads a memory location before the first epoch stores to that memory location. A backwards dependence causes a violation., 14

Buffer pool: The buffer pool is the subset of the database pages which 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., 45

Chain violation: When an epoch is restarted due to a violation it forces all later epochs to restart as well through a chain violation. This is because the later epochs may have consumed incorrect speculative values generated by the violated epoch., 14

Clean replica: if replication has created multiple versions of a cache line, the clean replica is the version with no speculative modifications., 69

Cursor: 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., 33

Data dependence: When a load is executed it expects to receive the value placed in the memory location by the previous store in the original program order. The relationship between the load and the previous store is a *data dependence*. TLS re-orders the execution of the program, and must ensure that data dependences are preserved to preserve correct execution., 13

Epoch context: The hardware which stores the speculative state for a single epoch (or sub-epoch)., 67

Epoch: The parallel unit of work performed by a speculative thread. An epoch either commits its work to memory or violates (undoes its execution)., 3

Escaping speculation: Temporarily disabling the speculation mechanism while the epoch executes to avoid dependences. While the CPU is in escaped mode all loads and stores are performed as if the epoch is homefree. The software is responsible for undoing the effects of escaped code if the epoch is later violated and restarted., 24

Exposed load table: A table which tracks which PC performed the first load to each memory address accessed by an epoch., 72

Exposed load: The first load to any particular memory location performed by an epoch. When a data dependence violation is detected then the consuming epoch must rewind execution *at least* to the exposed load which caused the violation., 72

False sharing: Data dependences in TLS are tracked at a *cache line* granularity. If two variables happen to reside on the same cache line, then a write to one variable may cause a violation if the other variable is read. This is a *false* dependence, caused by *false sharing*. The most common fix for this problem is to re-arrange the variables, or to insert padding between shared variables to ensure they reside on separate cache lines., 48

Forwarding frame: The portion of the stack used by TLS to pass initial arguments from one epoch to the next., 20

Homefree token: A token passed from one epoch to the next to indicate that the epoch can no longer be violated by prior epochs, or is homefree., 21

Homefree: The oldest epoch is said to be *homefree*, since there are no prior epochs which can cause it to be violated. An epoch becomes homefree when it receives the *homefree token* from the previous epoch., 21

Invalidation required buffer (IRB): A list of cache lines which have been replicated. At epoch commit time invalidations may have to be sent out for each of these lines to remove stale replicas from the L1 caches., 70

Isolated undoable operation: A function which has a corresponding undo function (is undoable), and if that undo function is invoked no other transaction or earlier epoch is forced to rewind or otherwise alter its execution., 27

Last-touch predictor: Hardware which attempts to predict the last time an epoch writes to a memory location., 66

Latch: A latch provides mutual exclusion between two threads. Latch is a database systems term, a latch is known as a *mutex* in operating systems parlance., 1

Latency: The latency of a transaction is how long it takes from the start of the transaction's execution until its completion., 49

Lock: A lock is used to allow safe concurrent access to an object by multiple threads. 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., 44

Ownership required buffer (ORB): A list of cache lines which have been speculatively modified but are not owned exclusively by a cache. When an epoch commits it sends out invalidations for all lines listed in the ORB to ensure that it has the only copy of a line before committing speculative changes., 70

Post-commit region: The region of an epoch's execution after the epoch has committed and passed the homefree token. Code which causes violations but is safe to be executed in parallel can be delayed until this region of the epoch., 24

Post-homefree region: The region of an epoch's execution after the homefree token has arrived but before the token is passed on to the next epoch. Code which frequently causes violations can be moved down to this region., 21

Pre-fork region: The region of an epoch's execution where the loop index is computed before starting the following epoch., 21

Primary violation: A violation caused by reading an incorrect speculative value. When an epoch is restarted due to a primary violation, it restarts all later epochs with a *chain violation*., 77

Recovery function: A user-specified function which is invoked when an epoch is violated. This is used when *escaping speculation* to ensure that the system state is properly restored if a violation occurs., 73

Replica conflict: A replica conflict is when two epochs need to keep different versions of a cache line in order to make forward progress. The TLS hardware presented in this thesis will *replicate* the cache line and maintain two versions to avoid a replica conflict., 68

Replication: (of a cache line) Making a duplicate version of a cache line to avoid a *replica conflict*., 67

Speculative region: The main body of an epoch; the region of an epoch's execution where the TLS mechanism ensures that execution is equivalent to the original sequential execution., 21

Speculative victim cache: A victim cache is a small fully-associative cache which stores cache lines which have been recently evicted from the main cache. A *speculative* victim cache is a victim cache which only stores speculative lines. The purpose of the speculative victim cache is to avoid violations caused by evicting speculative state., 68

Sub-epoch start table: This is a table which tracks the temporal relationship between sub-epochs. It is used to limit the amount of execution that is undone when an epoch receives a chain violation., 78

Sub-epoch: Epochs are broken up into sub-epochs to reduce the penalty of a violation. When a violation occurs the epoch is rewound to the start of the sub-epoch which contained the mis-speculated load, and not back to the start of the entire epoch. Sub-epochs can be thought of as a way to provide periodic checkpoints of the epoch's execution., 17

TPC-C: Transaction Processing Council benchmark C. This benchmark attempts to simulate the workload on a database server owned by a bank or business selling widgets. Further information and the benchmark specification can be found at `http://www.tpc.org`., 1

Thread: In this thesis 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., 27

Throughput: In a database context, throughput is the rate of transaction execution (transactions executed per minute)., 49

Violation: When TLS detects that an epoch has speculatively loaded an incorrect value, it generates a violation. The violation causes the epoch to rewind and restart. On the second execution the correct value will be consumed., 14