Supporting Large Speculative Threads for Databases and Beyond

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

July 2005
CMU-CS-05-109

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

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

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

Abstract

Thread level speculation (TLS) has proven to be a promising method of extracting parallelism from both integer and scientific workloads. In this paper we use TLS to exploit *intra-transaction* parallelism in database workloads, and demonstrate that previous hardware support for TLS is insufficient for the resulting large speculative threads (or *epochs*) and the complexity of the dependences between them. In this paper we extend previous TLS support in three ways to facilitate large epochs: (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-epoch 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.

1 Introduction

Now that the microprocessor industry has shifted its focus from simply increasing clock rate to increasing the number of processor cores integrated onto a single chip [2, 14, 16, 33], an increasingly important research question is how can we make it easier to take full advantage of these computational resources. One potential answer to this question is *Thread-Level Speculation* (TLS) [8, 10, 11, 17, 22, 23, 30, 34, 39], which enables either a compiler [36, 38] or the programmer [4, 24] to optimistically and incrementally introduce parallelism into a program while always maintaining functional correctness.

While *Thread-Level Speculation* (TLS) has been the subject of many recent research studies that have proposed a variety of different ways to implement its hardware and software support [8, 10, 11, 17, 22, 23, 30, 34, 39], a common theme in nearly all of the TLS studies to date is that they have focused on benchmark programs (e.g., from the SPEC suite [27]) where the TLS threads (or *epochs*) are of modest size: typically a few hundred to a few thousand instructions per epoch [24, 31, 36], or on benchmarks with very infrequent data dependences [6]. In addition, most of the mechanisms for supporting TLS that have been proposed to date take an *all-or-nothing* approach in the sense that a given epoch only succeeds in improving overall performance if it suffers *zero* inter-epoch dependence violations. (Otherwise, the epoch is restarted at the beginning, which hopefully does not hurt overall performance, but is not likely to improve it other than through prefetching effects.) While an all-or-nothing approach may make sense for the modest-sized epochs found in SPEC benchmarks where inter-epoch dependences are relatively infrequent, this paper explores a far more challenging (and possibly more realistic) scenario: how to effectively support TLS when it is applied to a *database management system* (DBMS) running a transaction-processing workload where the epoch sizes are much larger and where the complexity of the epochs causes inter-epoch dependence violations to occur far more frequently.

1.1 Using Thread-Level Speculation to Parallelize Individual Database Transactions: A Success Story that Motivates New Architectural Support

The traditional approach to exploiting parallelism within a transaction-processing workload has been to run separate transactions on separate processors. In a recent paper [4], however, we demonstrated that TLS can be successfully used to parallelize *individual* transactions. By exploiting this complementary form of parallelism, we can reduce the *latency* of individual transactions (rather than having to settle only for improvements in overall transaction-processing throughput).

Our experience with parallelizing individual database transactions [4] was a major success story for TLS for the following reasons. First, database management systems (DBMSs) are very large, complex pieces of code. For example, the BerkeleyDB [21] system that we worked with contains roughly 180,000 lines of code. Second, BerkeleyDB and commercial DBMSs have been written under the explicit assumption that there will *never* be parallel execution *within* a transaction (as opposed to across separate transactions): they take full advantage of this assumption in the internal DBMS structures and routines. Hence rewriting these very large software systems to make traditional parallel execution safe within a given transaction would involve a prohibitively

large amount of programmer effort. By exploiting TLS support, however, a graduate student who was unfamiliar with the BerkeleyDB code base was able to incrementally introduce TLS-style parallelism by hand with less than a month's worth of effort (changing less than 1200 out of the 180,000 lines of code), ultimately resulting in roughly a *twofold speedup* on a simulated 4-way chip multiprocessor for the most important transaction in TPC-C [9] (NEW ORDER), as illustrated later in Section 5. Success stories like these were exactly the motivation for TLS.

To achieve this success, however, we first needed to overcome some new challenges that did not arise in studies of smaller programs such as the SPEC [27] 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.

1.2 Related Work

The idea of using speculative execution to simplify manual parallelization is a fairly recent one, first proposed by Prabhu and Olukotun for parallelizing SPEC benchmarks on the Hydra multiprocessor [10, 24]. The base Hydra multiprocessor uses a design similar to the L2 cache design proposed in this paper—the design in this paper extends the Hydra design by adding support for mixing speculative and non-speculative work in a single epoch, as well as allowing partial rollback of a epoch through sub-epochs. Hammond *et. al.* pushes the idea of programming with epochs to its logical extreme, making the program consist of nothing but epochs (transactions), resulting in a vastly simpler architecture [13], but requires changes to the programming model to accommodate the new architecture [12]. The architecture presented in this paper is closer to existing architectures, and can run today's executables as well as executables which have been modified to take advantage of TLS.

Sub-epochs are a form of checkpointing, and in this paper sub-epochs are used to reduce the penalty due to failed speculation. Prior work has used checkpointing to simulate an enlarged reorder buffer with multiple checkpoints in the load/store queue [1, 5], and a single checkpoint in the cache [19]. Martínez's checkpointing scheme [19] effectively enlarges the reorder buffer and is also integrated with TLS, and is thus the most closely related work. The sub-epoch design in this paper could be used to provide a superset of the features in Martínez's work at a higher cost: sub-epochs could provide multiple checkpoints with a large amount of state in a cache shared

by multiple CPUs. 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 [35]—the techniques used to create sub-epochs in this paper could also be used to create checkpoints at high-confidence prediction points using the techniques from Tuck's paper. In this paper we use sub-epochs to tolerate data dependences between speculative epochs; other studies have explored predicting data dependences and turning them into synchronization [20, 31], or have used the compiler to mark likely dependent loads and tried to predict the consumed values at run time [31]. Initially we tried to use an aggressive dependence predictor like proposed by moshovos [20], but found that only one of several instances of the same load PC caused the dependence—predicting which instance of a load PC is more difficult, since you need to consider the outer calling context. Sub-epochs provides a more elegant solution which is *complementary* to hardware based prediction and software based synchronization techniques, since using sub-epochs significantly reduces the high cost of mis-speculation.

1.3 Contributions

The primary contribution of this paper is that it introduces sub-epochs, a mechanism to tolerate data dependences between large epochs. This allows us to investigate applying TLS to commercial database transactions, which has not been examined in the past. We believe that the techniques applied in this paper can be applied to make large epochs more effective in other benchmarks as well.

To parallelize these large dependent epochs we needed more than just sub-epoch support—we needed to have a TLS system which could buffer large epochs, tolerate forward dependences between epochs (through aggressive update propagation), provide hardware mechanisms to profile data dependences, and provide mechanisms to help programmers remove critical dependences from software. These individual pieces have been partially or fully demonstrated in previous papers, in this work we present a unified design which incorporates all of these features together to deal with database transactions.

This paper is organized as follows: In Section 2 we describe a TLS architecture which builds upon prior TLS designs to allow the buffering of very large epochs. Section 3 shows how hardware support for profiling violations and escaping the speculation mechanism allows the DBMS to be *tuned* to remove the most performance critical data dependences. Section 4 introduces *sub-epochs*, a form of checkpointing for speculative epochs, which allows the TLS execution to *tolerate* data dependences, and minimizes the amount of execution rewound when a dependence is detected. Section 5 presents results, and in Section 6 we conclude.

2 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

Cache line conflict: If two epochs need to store speculative state in a cache line but they can not both store that information without losing some information, it is called a cache line conflict. One solution to this problem is to replicate the cache line.

Clean replica: A replica of a cache line with no speculative modifications.

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

Escape speculation: When an epoch needs to make non-speculative changes to memory it escapes speculation to do so. When it is finished, it resumes speculation. Great care must be taken when escaping speculation, as the escaped code may have been invoked by a mis-speculating epoch.

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.

Implicit forwarding: Allowing an epoch to view the writes performed by prior epochs as soon as they happen, instead of waiting until the prior epoch commits.

Invalidation required buffer: A list of cache lines (tags) which require invalidations to be sent out and cache line replicas to be merged when the epoch commits.

Isolated undoable operation (IUO): An operation is *isolated* if rewinding its execution does not require other epochs to rewind, and *undoable* if there is a corresponding *undo* operation which returns the program to the same logical state it was in before the original operation was performed. TLS takes advantage of this by performing the operation as a single unit (as viewed by the TLS mechanism), using the *escape speculation* mechanism. If the epoch is violated then the corresponding undo operation is invoked.

Replica: A copy of a cache line used to resolve a cache line conflict. Also called a *version* in other papers.

Speculative victim cache: A victim cache which can also store (and track) speculative state. Used to compensate for the limited associativity of the cache.

Sub-epoch: A fragment of an epoch. If the TLS hardware tracks work done by sub-epochs it can decrease the amount of work that is undone by a violation.

Violation recovery function: The function which must be invoked on a violation to undo the changes made by an IUO.

Figure 1: Glossary of terms used in this paper.

(a) Sequential execution.

(b) Parallel execution with TLS.

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

the offending epoch (Figure 2). Hardware support for TLS makes the detection of violations and the restarting of epochs inexpensive [10, 25, 26, 30].

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 [36], 140–7735 dynamic instructions [24], 30.8–2,252.7 dynamic instructions [31], and up to 3,900–103,300 dynamic instructions [6]. The epochs studied in this paper 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 [10, 26] or in the L1 cache [3, 8, 30]. 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 [25]; 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 younger epochs running on a CMP be restricted to the L1 cache) [3]. In this paper, 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 *sub-epochs*, described in later in Section 4.

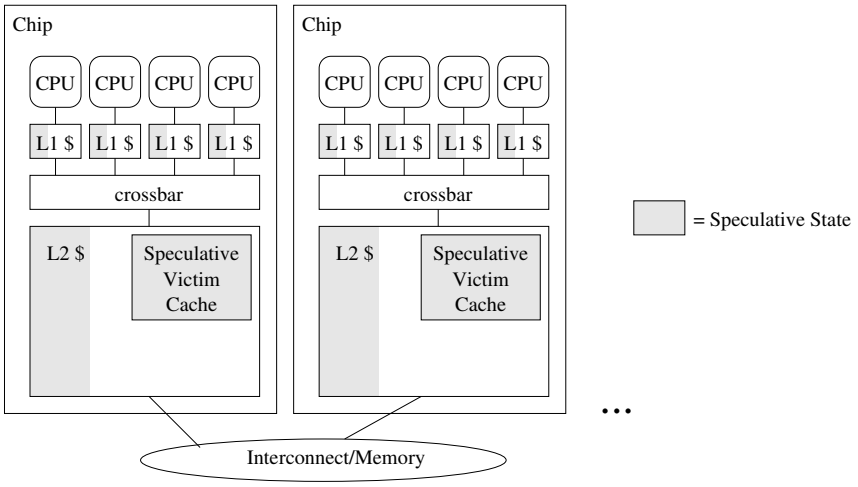


Figure 3: An overview of the CMP architecture that we target, and how it is extended to support TLS.

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

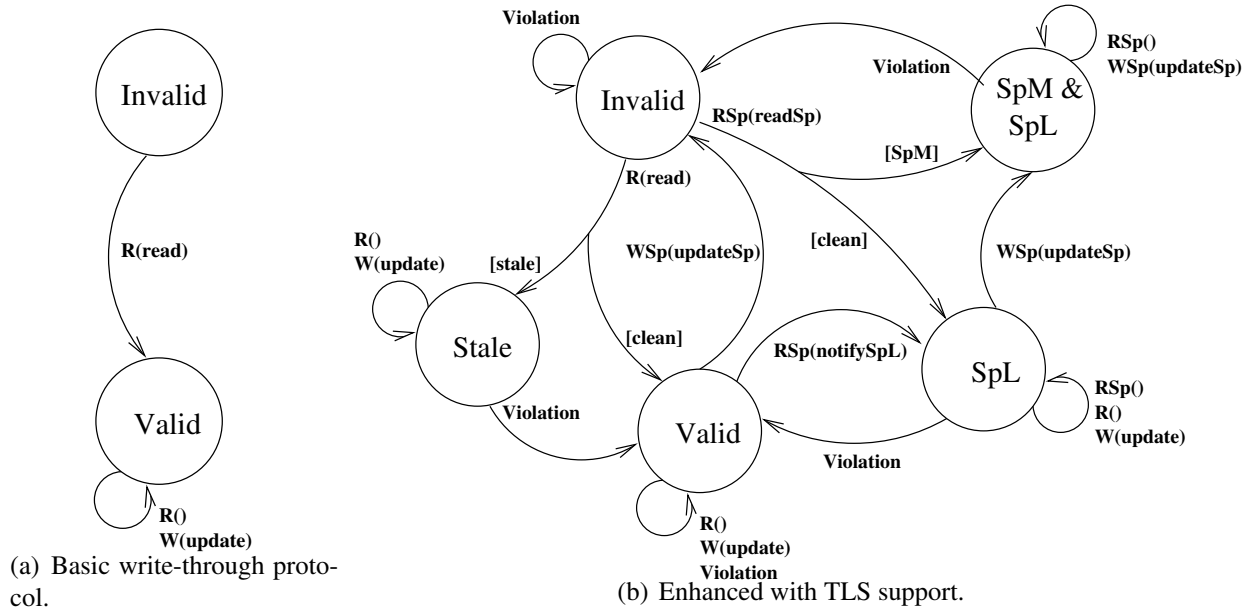
In this section we describe the underlying chip multiprocessor (CMP) architecture and how we extend it to handle large epochs. We assume a CMP where each core has a private L1 cache, and multiple cores share a single chip-wide L2 cache (Figure 3). For simplicity, in this paper each CPU executes a single 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 [30].[1]

2.1.1 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 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

[1] In this paper, 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: 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. Edges are labeled with the processor action, followed by the message to the L2 cache in round brackets. Square brackets show responses from the L2 cache.

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 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 5.5 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* [18] 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 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 [28].

The *Stale* state in Figure 4 is used for temporarily escaping speculation, and is discussed further

in Section 3.3.

2.1.2 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 [28]. 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 tracked at lookup by examining the L2 cache state, and dependences between epochs running on different L2 caches are tracked using the extended cache coherence proposed in prior work [28].

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 [8]; we present our replication scheme in detail in the next section to illustrate an alternate implementation, and since we build upon our replication scheme to implement sub-epochs in Section 4.

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* [15, 32] 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 [25] to store the overflowed speculative state.

2.1.3 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 "the future". 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*

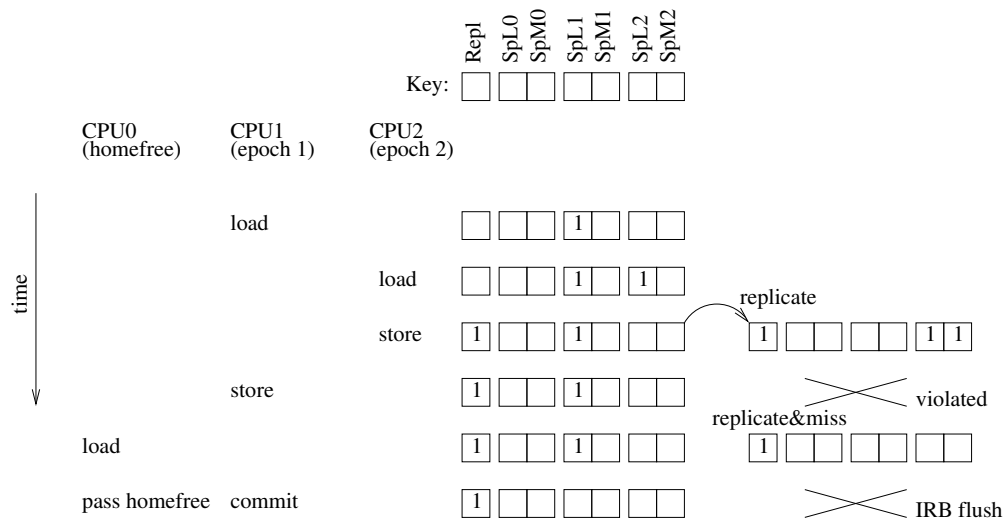


Figure 5: Replication in action.

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 to 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 5, Appendix A). 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. 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 neither the current epoch nor 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 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. 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.

2.1.4 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

[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. Alternatively, we could pin the clean replicas of cache lines in the cache until all replicas of a line are merged, but this wastes cache space.

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 [15], 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.

3 Using TLS to Incrementally Parallelize Database Transactions

We have applied TLS to the loops of the transactions in TPC-C and found that the resulting epochs 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). The many data dependences between epochs form a critical performance bottleneck. Previous work [4] has demonstrated 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.

3.1 Tuning the Database System

To evaluate our ideas we apply TLS to an existing database system, BerkeleyDB [21], running transactions from a TPC-C [9] like workload.[3] We have implemented the five transactions from TPC-C, namely NEW ORDER, DELIVERY, STOCK LEVEL, PAYMENT and ORDER STATUS—source code for our transaction implementation can be found in Appendix B.

The NEW ORDER transaction is the most frequently executed transaction in the TPC-C workload, as it is roughly 45% of the TPC-C transaction mix. This transaction models a customer order being fulfilled by a company with items in multiple warehouses. The transaction is structured as shown in Figure 6. After creating a new order record in the database, the transaction goes into a loop locating each item to be purchased and decrementing its count (the TPC-C specification says that each order will contain between 5 and 15 items). If all of the items are found, then the transaction commits and all of the changes are permanently recorded. For 1% of transactions an item will not be found, and the transaction aborts.

The bulk of the work in this transaction (87% of dynamic instructions) is done in the `for` loop. Usually each iteration of this loop will access a different item in the database, but this is not guaranteed. This makes the loop an ideal candidate for speculative parallelization—we use TLS to run the iterations of the loop in parallel, and let the TLS mechanism cause the iterations to be serialized in the hopefully rare case of a data dependence between loop iterations.

Considering just the operations in the transaction code it looks like the loop in Figure 6 is usually parallel. The only operation performed by this loop which may conflict between iterations is the decrement of the item quantity in the stock table: since items are chosen uniformly from a table of size 100,000, conflicts actually occur with very low frequency. Unfortunately, each operation performed by the transaction calls into the database system, and the database system executes a large amount of complex code in order to properly implement these operations. When run, the database system code causes numerous read-after-write dependence violations between the epochs, which hinder performance. To get good performance we have to parallelize the sections of the the database system code where frequent data dependences limit performance. Note that the database system is quite complex: BerkeleyDB has 175k lines of code—fortunately we only have to modify a tiny fraction of the code to achieve good performance.

[3] 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 paper 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 times portions of the benchmark. Because of this, the performance numbers in this paper should not be treated as actual TPM-C results, but instead should be treated as representative transactions.

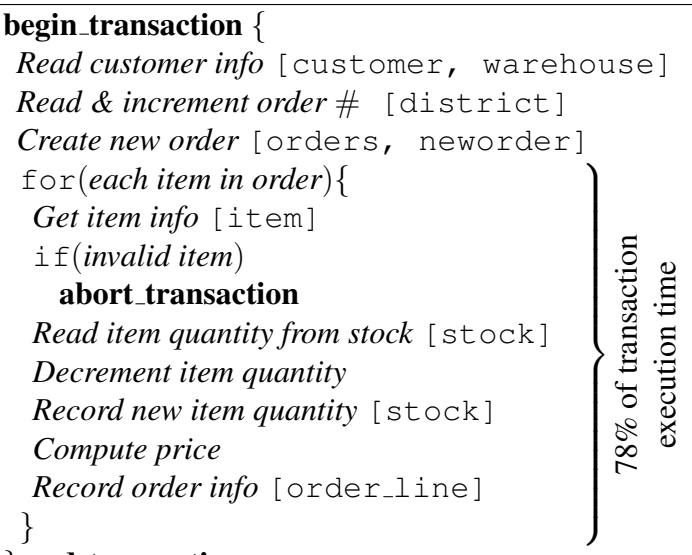


Figure 6: The NEW ORDER transaction. In brackets are the database tables touched by each operation.

In this paper we focus on the hardware and system support required for performance once the software has already been transformed (for more detail on the software transformations, see our previous paper [4]). The transformations are important but not overly complex: we only touched 42 of the 378 source files in BerkeleyDB, and the coding was performed by a graduate student who was learning database systems and also redesigning the TLS hardware as he coded, in under three months. The software transformations touch on all of the major software mechanisms in the database system. The structures modified are:

Latches: Latches enforce mutual exclusion between transactions, and are used quite frequently (database latches are the same as "mutexes" as taught in operating systems classes.) Acquiring and releasing a latch reads and writes a memory location, and these frequent reads and writes to the small set of memory locations used by latches causes violations between epochs. These violations are caused by the TLS mechanism attempting to preserve sequential semantics between epochs, while latches only need to preserve mutual exclusion *between transactions*. We modify the latch acquire and release operations to avoid violations and still preserve mutual exclusion between transactions.

We avoid violations and preserve mutual exclusion with two steps: 1. delay all mutex acquires until after the homefree token has arrived (and hence no violations can be caused by earlier epochs) and before the epoch commits (so when writes become visible to other transactions they are done while the mutex is held); and 2. delay all mutex releases until after the epoch commits (so other transactions do not observe any partially committed results without the protection of the mutex).

Locks: Locks are modified in a similar fashion to mutexes. Read-only (existence) locks are further

optimized: since no changes are made to the state guarded by the lock, and since TLS ensures that an epoch observes the system memory state at a single logical point in time, read-only locks can be acquired and released at any point during the epoch's execution before the epoch commits.

Cursor management: BerkeleyDB maintains a list of free *cursor* objects to use for searching B-trees. The management of this single list causes violations between epochs, so we partition this into one list per CPU.

Log management: Every operation performed by the database system on behalf of queries involves appending to the database log. These append operations cause violations between epochs, and cannot be easily parallelized since log sequence numbers must be generated serially. Instead of parallelizing the log system (a fundamental change to the database system), we queue up all log writes performed by an epoch and append them all in a batch after the homefree token has arrived and before the epoch commits.

Counters: Several global statistics gathering counters were replaced with per-CPU counters to avoid violations when counters are incremented or decremented.

Error checks: A couple of dependence causing error checks were removed.

False sharing: Since TLS tracks data dependences at a cache-line granularity, false sharing can cause violations. To avoid this performance hit padding was introduced to data structures which suffered from false sharing violations.

Resource allocation: The database system makes heavy use of memory allocators (such as `db_malloc` and `db_free`) for temporary objects. The page cache subsystem manages moving data pages between the disk and memory, and it is called using a `pin_page` and `unpin_page` interface. The implementation of both of these operations cause violations between epochs. We have found that we can apply wrappers to these operations to turn them into *isolated undoable operations* (IUOs) to avoid these violations. IUOs take advantage of escaping speculation, which is described further in Section 3.3.

To support this iterative parallelization process we require two capabilities from the underlying hardware. First, we need the hardware to provide profile feedback reporting which load/store pairs triggered the most harmful violations—those that caused the largest amount of failed speculation. Second, we need the hardware to allow the programmer to *escape* the speculation mechanism and run recovery handlers if a violation occurs. There are a small set of operations performed by the DBMS which are difficult to modify to avoid violations. Instead of executing those routines speculatively, we found it was simpler to execute those routines non-speculatively and undo their side-effects in software if a violation occurs.

3.2 Profiling Violated Inter-Epoch Dependences

To track which load and store PC pairs cause the most harmful dependence violations we could use a simulator or a software instrumentation pass. However, hardware support for such profiling

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

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* [31]—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 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.

3.3 Hardware Support for Escaping Speculation

It is easiest to explain escaping speculation with an example. Consider the code in Figure 7. 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 mis-speculation. 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* [7, 29]).[4] 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.

3.4 Impact of Performance Tuning

In Figure 8 we see the performance impact of each round of the optimization process on the latency of the NEW ORDER transaction: the SEQUENTIAL bar shows the performance of the unparallelized

[4]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.

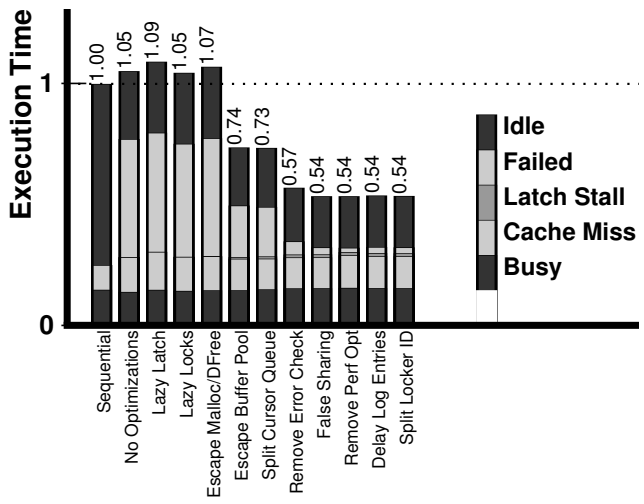
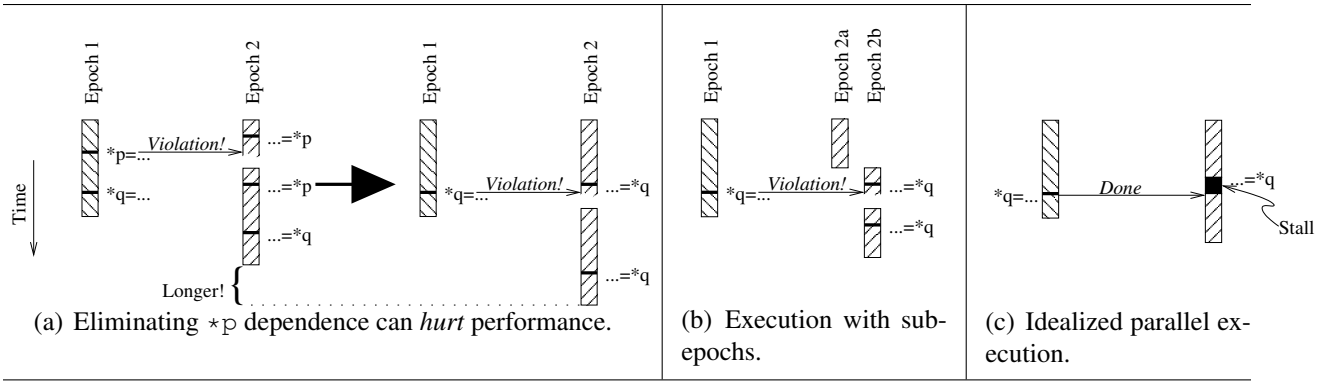


Figure 8: Performance impact on NEW ORDER of adding each optimization one-by-one on a four CPU machine.

transaction run on one CPU, and the NO OPTIMIZATIONS bar shows the performance of the TLS-parallelized benchmark on four CPUs before any tuning has been done. Each bar to the right of the NO OPTIMIZATIONS bar shows the improvement caused by eliminating the dependence which causes the worst performance bottleneck; each bar adds an optimization to the bar to its left. The first few transformations add software overhead but do not uncover enough parallelism to improve performance. The first major performance improvement occurs once the ESCAPE BUFFER POOL optimization is applied—this optimization (and the ESCAPE MALLOC/DFREE optimization) takes advantage of the ability of the system to escape speculation. The performance counters report the amount of time spent on failed speculation, so the programmer can easily bound any performance improvements achieved by further tuning: in Figure 8 the fraction of execution time spent on failed speculation is represented by the FAILED segment of the bars, and it is apparent that further optimization will not be productive.

4 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 programmer to eliminate dependences one-by-one; but eliminating one dependence may expose an even-later dependence, which can potentially make performance *worse* (Figure 9(a)). Previous work has proposed data dependence predictors and value predictors to tolerate inter-epoch dependences [20, 31]: 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),



(a) Eliminating $\star$p dependence can *hurt* performance.

(b) Execution with sub-epochs.

(c) Idealized parallel execution.

Figure 9: Sub-epochs improve performance when dependences exist.

hence any predictor would have to be very accurate to avoid violations.

In this paper we do not rely on predictors to avoid violations—instead, we reduce the cost of violations by using *sub-epochs.* 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 9(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 9(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 communicates values between epochs. In Figure 10, 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.

4.1 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.

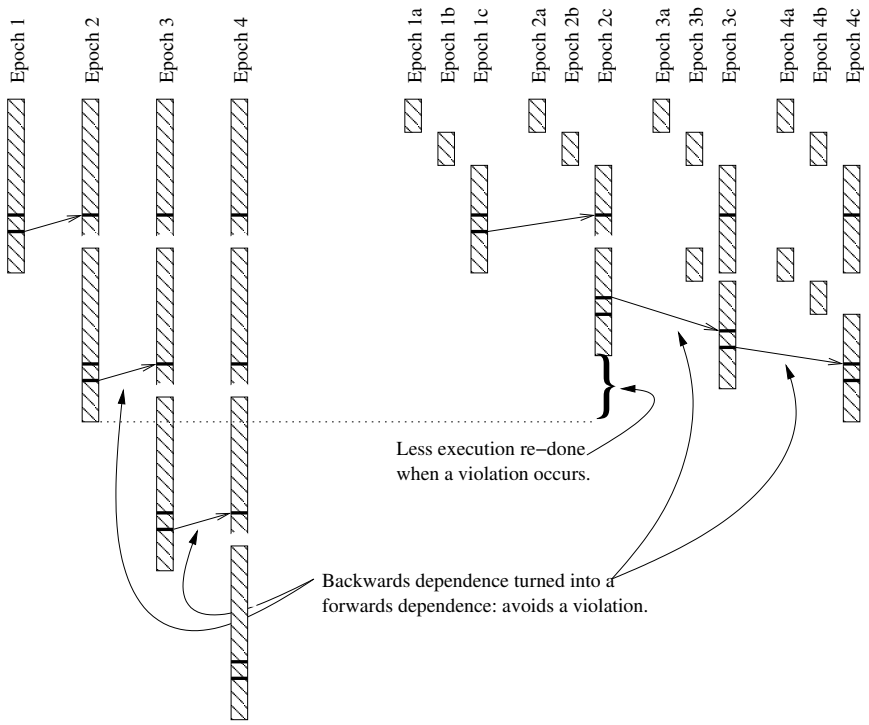


Figure 10: The two sources of performance improvement from sub-epochs.

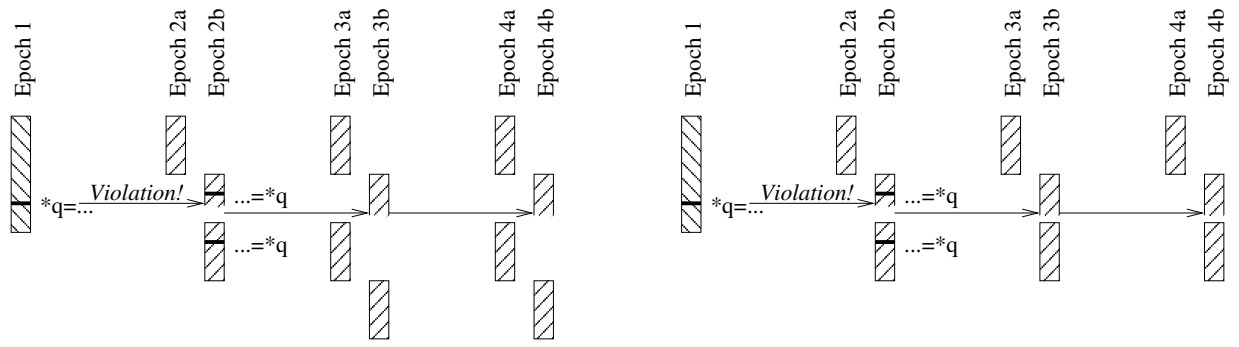


Figure 11: The effect of secondary violations with and without sub-epoch dependence tracking.

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 *secondary violation*. With sub-epochs this behavior is suboptimal, as illustrated in Figure 11(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 11(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.

4.1.1 Sub-epoch Start Table

To ensure that secondary 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 secondary 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.[5] 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:

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

4.2 Choosing Sub-epoch Boundaries

Using sub-epochs limits the amount of execution rewound on a mis-speculation; but as a given epoch executes, when should the system start each new sub-epoch? How many sub-epochs are necessary for good performance? Sub-epochs have a hardware overhead, so supporting only a small number of sub-epochs per epoch is preferable—although the system will have to use them sparingly.

Since inter-epoch dependences (and hence violations) are rooted at loads from memory, we want to start new sub-epochs just before certain loads. 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 [31]. 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 5.4.

5 Experimental Results

In this section we assume that the DBMS programmer has created a TLS-parallel version of TPC-C, and use it to evaluate the hardware design decisions presented in this paper. For further information on how the programmer achieves this, see our previous paper [4].

[5] 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.

5.1 Benchmark Infrastructure

Our experimental workload is composed of the five transactions from TPC-C (NEW ORDER, DELIVERY, STOCK LEVEL, PAYMENT and ORDER STATUS).[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 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. Also, since we are primarily concerned with parallelism at the CPU level, we attempt to avoid I/O by configuring 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. In a previous paper [4] we described how we iteratively optimized the database system for TLS using the methodology described in Section 3. In this section we evaluate the hardware using the fully optimized benchmark.

5.2 Simulation Infrastructure

We perform our evaluation using a detailed, trace-driven simulation of a chip-multiprocessor composed of 4-way issue, out-of-order, superscalar processors similar to the MIPS R14000 [37], 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. The second level cache has a 64-entry speculative victim cache which holds speculative cache lines that have been evicted due to conflict misses. Register renaming, the reorder buffer, branch prediction, instruction fetching, branching penalties, and the memory hierarchy (including bandwidth

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

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

Table 1: Simulation parameters.

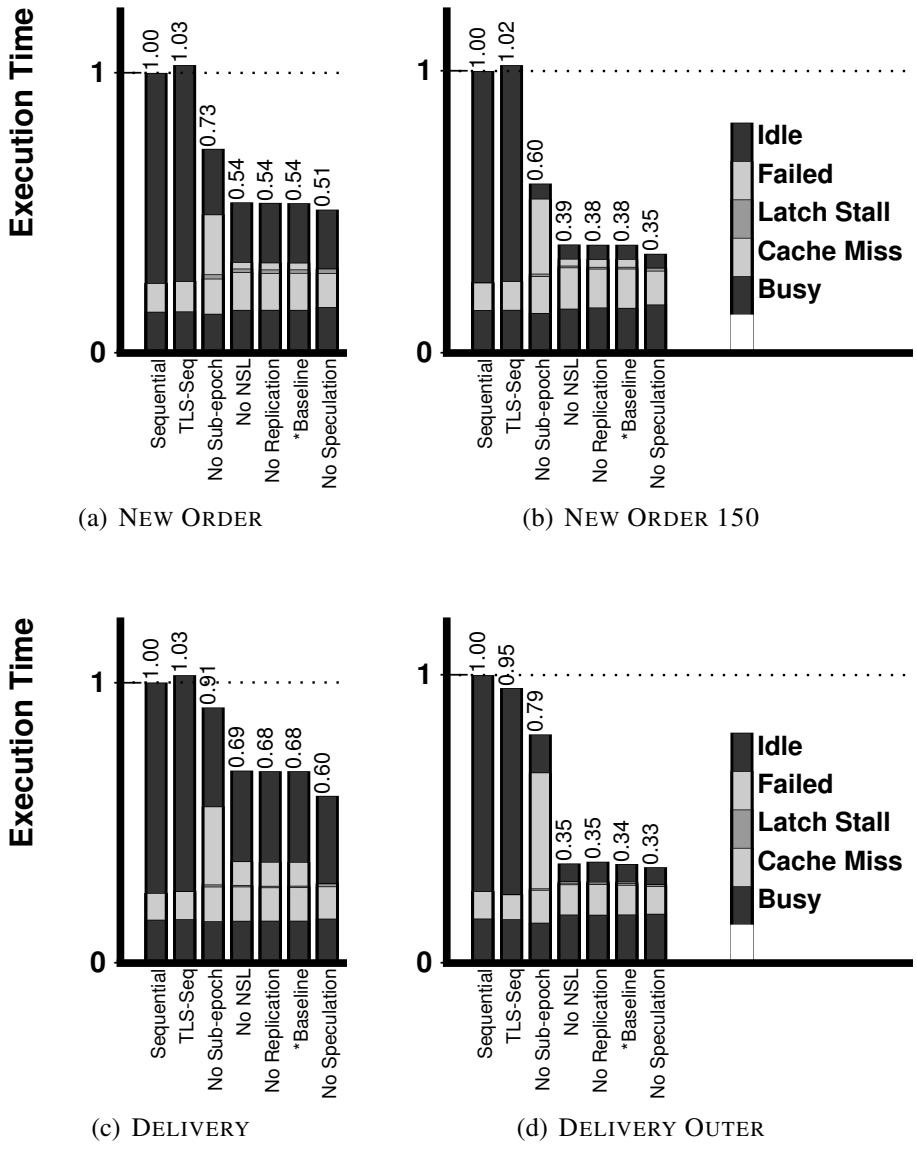
Pipeline Parameters	
Issue Width	4
Functional Units	2 Int, 2 FP, 1 Mem, 1 Branch
Reorder Buffer Size	128
Integer Multiply	12 cycles
Integer Divide	76 cycles
All Other Integer	1 cycle
FP Divide	15 cycles
FP Square Root	20 cycles
All Other FP	2 cycles
Branch Prediction	GShare (16KB, 8 history bits)

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

and contention) are all modeled, and are parameterized as shown in Table 1. Latencies due to disk accesses are not modeled, and hence these results are most readily applicable to situations where the database's working set fits into main memory.

5.3 High Level Benchmark Characterization

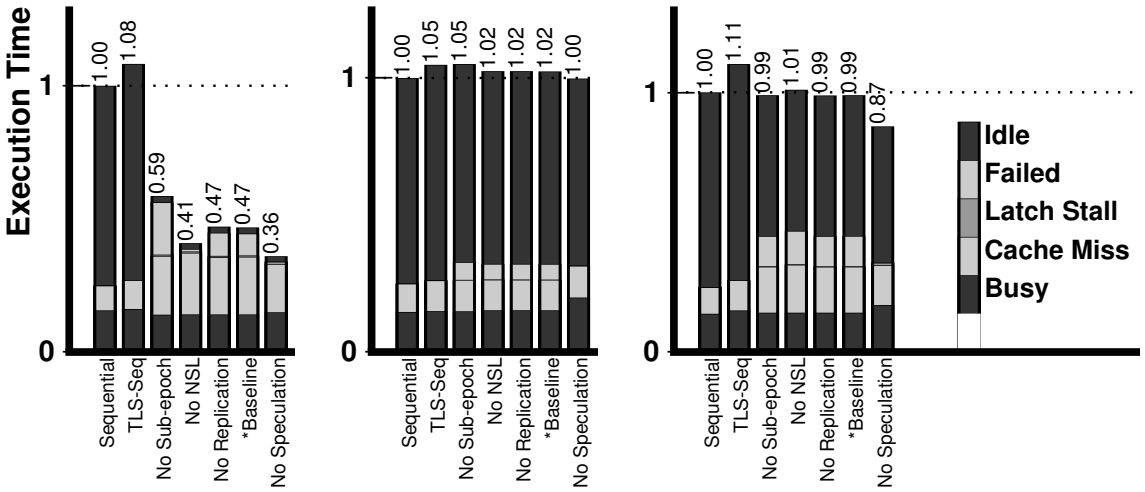
We start by characterizing the benchmark itself, so we can better understand it. As a starting point for comparison, we run our original sequential benchmark, which shows the execution time with no TLS instructions or any other software transformations running on one CPU of the machine (which is configured with 4 CPUs, cache line replication, and sub-epoch support enabled). This SEQUENTIAL experiment takes between 17 and 509 million cycles (Table 3) to execute, but this time is normalized to 1.0 in Figure 12. (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 12 shows the performance of this parallelized executable running on a single CPU—the



(a) NEW ORDER (b) NEW ORDER 150

(c) DELIVERY (d) DELIVERY OUTER

Figure 12: Overall performance of optimized benchmark.



(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 NSL	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 paper.
No Spec	Upper bound—modified hardware to treat all speculative writes as non-speculative.

Figure 12: *Continued.*

Table 2: Explanation of graph breakdown.

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

Table 3: 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

additional software overhead is reasonable, varying from -5% to 8% (negative overheads are due to our added code inadvertently improving the performance of the compiler optimizer).

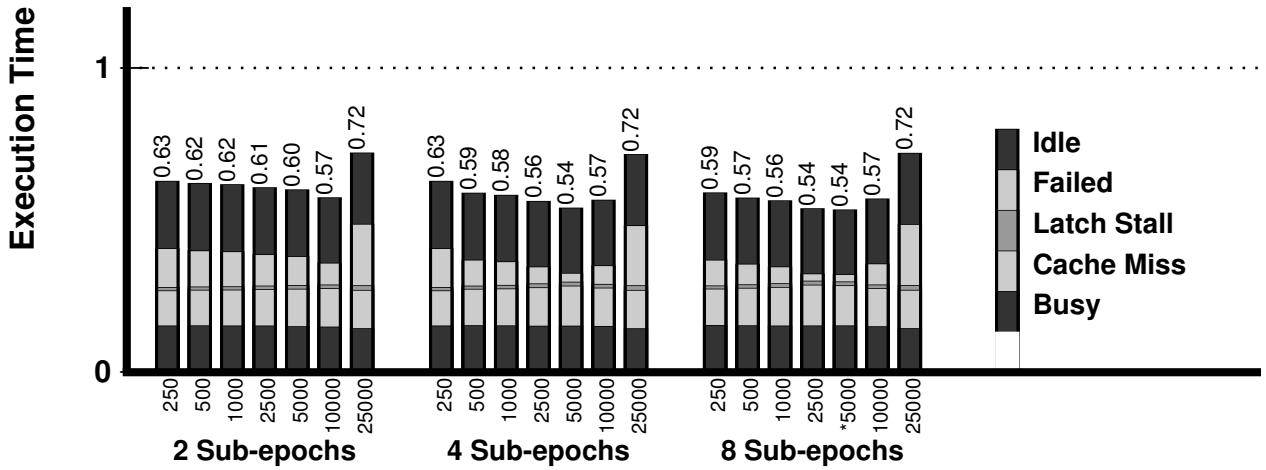
When we apply TLS with our BASELINE hardware configuration (4 CPUs, 8 sub-epochs per epoch, 5000 speculative instructions per sub-epoch) we see a significant performance improvement for three of the five transactions, with a 46%–66% reduction in execution time. The PAYMENT and ORDER STATUS transactions do not improve with TLS, and so we omit them from further discussion.

Is it possible to do better? In the NOSPEC 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 the ORDER LINE table. The STOCK LEVEL transaction is limited by dependences on the cursor used to scan the ORDER LINE table.

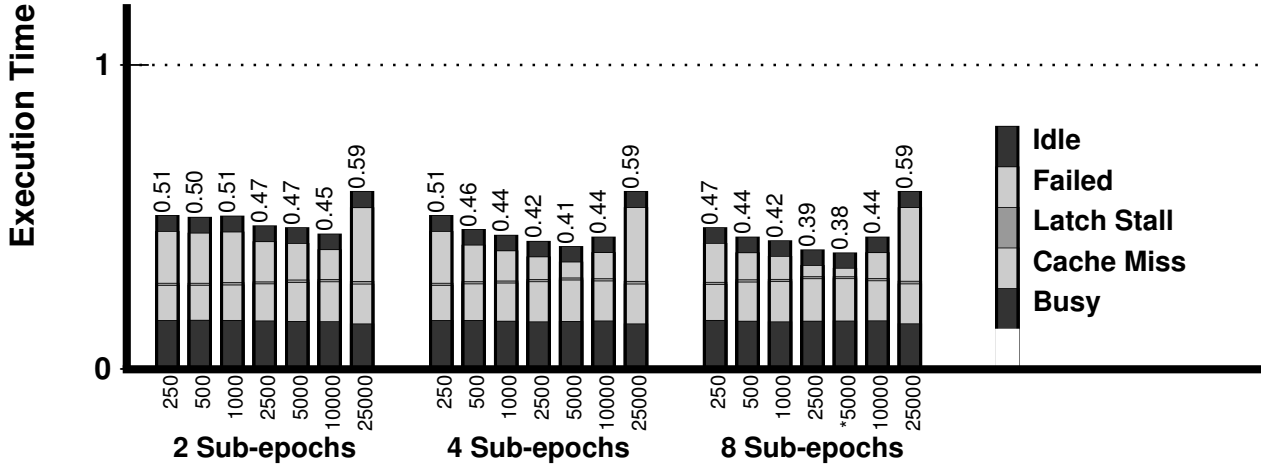
In the following sections we try to learn how the various portions of our design contribute to these overall performance results.

5.4 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 12 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.

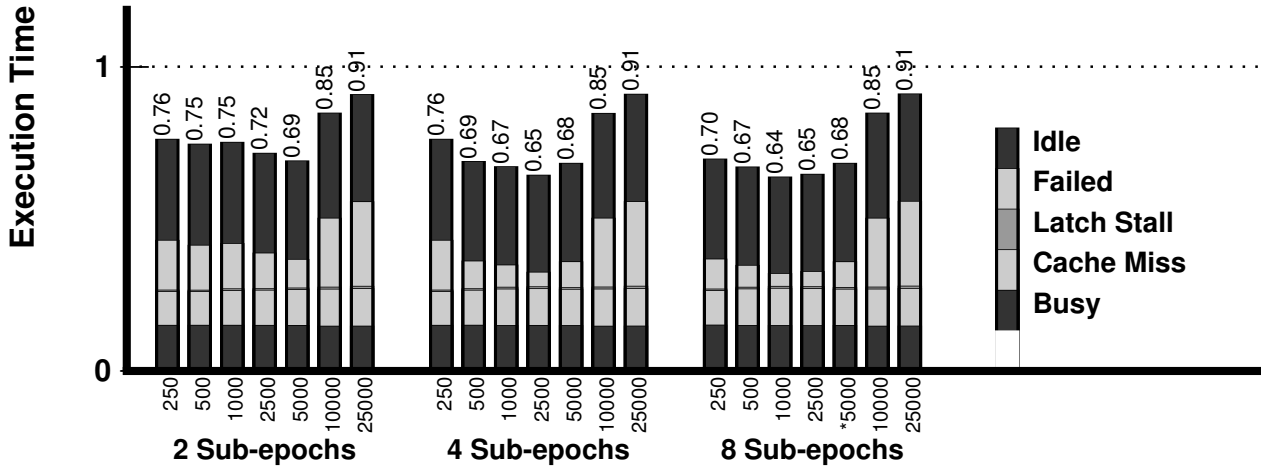


(a) NEW ORDER

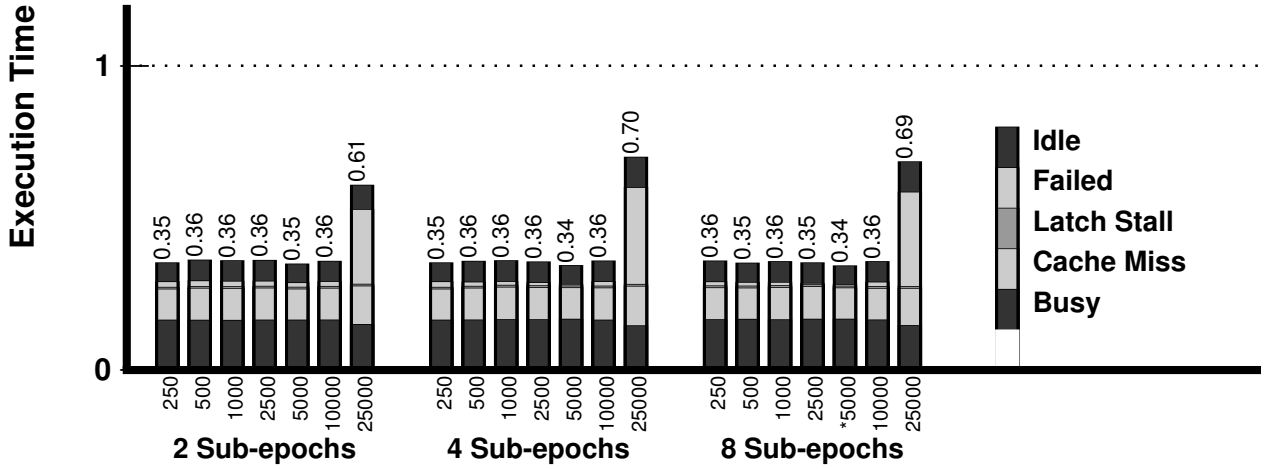


(b) NEW ORDER 150

Figure 13: 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.

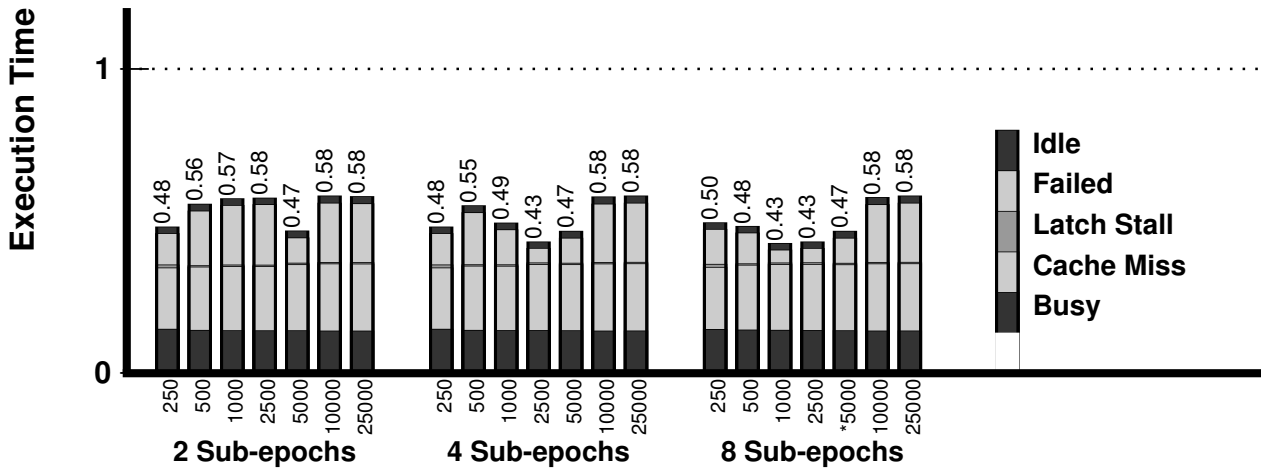


(c) DELIVERY



(d) DELIVERY OUTER

Figure 13: *Continued.*



(e) STOCK LEVEL

Figure 13: *Continued.*

5.4.1 Sub-epoch Size and Placement

How many sub-epochs are needed, and when should the system use them? In Figure 13 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 13 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).

When we initially chose a distance between sub-epochs of 5000 dynamic instructions it was somewhat arbitrary: we chose a round number which could cover most of the NEW ORDER transaction with 8 sub-epochs per epoch. This value has proven to work remarkably well for all of our

transactions. Closer inspection of both the epoch sizes listed in Table 3 and the graphs of Figure 13 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 14(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 14(d)) this secondary benefit of sub-epochs does not occur.

5.5 Cache Configuration

5.5.1 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 12 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 NSL bar in Figure 12 we remove this message, and cause a 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.

5.5.2 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 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

Table 4: 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

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.

5.6 Scaling Properties

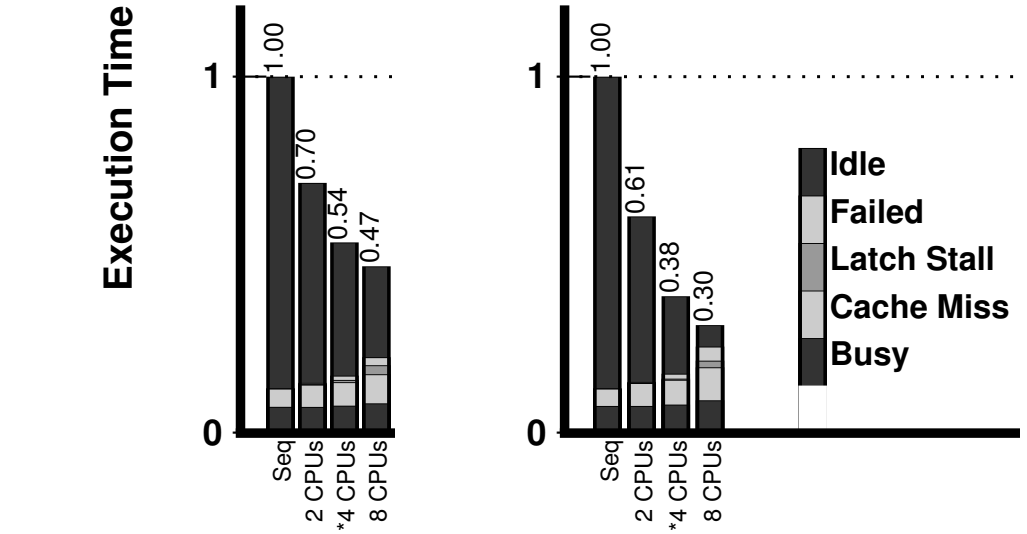
In Figure 14 we see the performance of the optimized transaction as the number of CPUs is varied.[8] The SEQUENTIAL bar represents the unmodified benchmark running on a single core of an 8 core chip multiprocessor, while the 2 CPU, 4 CPU and 8 CPU bars represent the execution of full TLS-optimized executables 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. This experiment assumes that the latency to the L2 cache does not change with the number of CPUs, which would be the case in a 8-CPU system where one had to decide how many CPUs to allocate to a particular transaction.

To better understand this data we break down each bar by where time is being spent—the breakdown is explained in Table 2. In Figure 14 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 7 CPUs idling, the 2 CPU breakdown shows two CPUs executing and 6 CPUs idling, and so on.

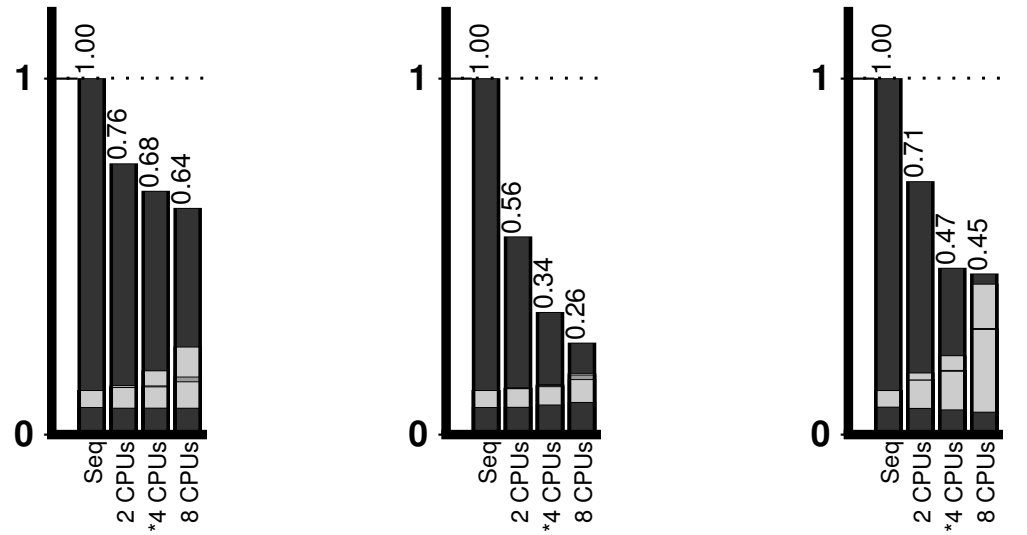
Most of the bars show that a significant fraction of the time was spent on failed speculation—this means that our performance tuning was successful at eliminating performance critical data dependences. As the number of CPUs increases there is a nominal increase in both failed speculation and time spent awaiting the mutex used to serialize portions of code executed with speculation escaped: as more epochs are executed concurrently, contention increases for both shared data and the mutex. As the number of CPUs increases there is also an increase in time spent awaiting cache misses: spreading the execution of the transaction over more CPUs decreases cache locality, since the execution is partitioned over more level 1 caches.

The dominant component of most of the bars in Figure 14 is *idle* time, for three reasons. First, in the SEQUENTIAL, 2 CPU and 4 CPU case we show the unused CPUs as idle. Second, the loops that we parallelized in the transactions have limited coverage, ranging from 78% to 99% (Table 3), and during the remaining execution time only one CPU is in use. Third, the parallelized loops

[8]The results in this section also appear in a previous paper which describes the optimization to the database software [4].



(a) NEW ORDER (b) NEW ORDER 150



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

Figure 14: Scaling the number of CPUs (4 CPUs is the baseline).

have a limited number of epochs available: for example, TPC-C specifies that the NEW ORDER transaction will order between 5 and 15 items, which means that on average each transaction will have only 10 epochs—this means that as we execute the last epochs in the loop load imbalance will leave CPUs idling. The effects of all three of these issues are magnified as more CPUs are added. To reduce idle time we modified the invocation of the NEW ORDER transaction so that each order contains between 50 and 150 items (shown as NEW ORDER 150 in Figure 14(b)). We found that this modification decreases the amount of time spent idling, and does not significantly affect the trends in cache usage, violations, or idle time. The modified bars demonstrate that transactions which contain more parallelism make more effective use of CPUs.

The results in Figure 14 show that there is a performance trade-off when using TLS to exploit intra-transaction parallelism: devoting more CPUs to executing a single transaction improves performance, but there are diminishing returns due to a lack of parallelism, increased contention, and/or a decrease in cache locality. One of the strengths of using TLS for intra-transaction parallelism is that it can be enabled or disabled at any time, and the number of CPUs can be dynamically tuned. The database system's scheduler can dynamically increase the number of CPUs available to a transaction if CPUs are idling, or to speed up a transaction which holds heavily contended locks. If many epochs are being violated, and thus the intra-transaction parallelism is providing little performance benefit, then the scheduler could reduce the number of CPUs available to the transaction. If the transaction compiler simply emitted a TLS parallel version of *all* loops in transactions then the scheduler could use sampling to choose loops to parallelize: the scheduler could periodically enable TLS for loops which are not already running in parallel, and periodically disable TLS for loops which are running in parallel. If the change improves performance then it is made permanent.

6 Conclusion

In this paper we have demonstrated that TLS can enable a database programmer to easily exploit *intra-transaction parallelism* to improve transaction latency, but that previously-proposed hardware support for TLS is insufficient for the resulting large speculative threads. We overcame these limitations with the following additional hardware support for large speculative threads: (i) a novel two-level cache protocol that allows a CMP with TLS support to fully exploit the large capacity of the L2 cache to buffer speculative state; (ii) support for aggressive update propagation to help tolerate inter-thread dependences; (iii) support for profiling violated inter-thread dependences that allows the programmer to track the most problematic load/store pairs, and then if possible to modify the corresponding database code to remove the dependence; (iv) support for temporarily escaping speculation, for operations such as `malloc` which can be easily undone if speculation fails; and (v) support for sub-threads which automatically breaks large speculative threads into smaller units, allowing a thread to commit or re-execute at a finer granularity—thereby tolerating unpredictable inter-thread dependences. We explored different possibilities for how to break speculative threads into sub-threads and found that having hardware support for eight sub-thread contexts, where each sub-thread executes roughly 5000 dynamic instructions, performed best on average. With this support we reduce transaction latency by 46–66% for three of the five TPC-C transactions considered, showing that TLS can provide exciting performance benefits to important

domains beyond general-purpose and scientific computing.

References

[1] Haitham Akkary, Ravi Rajwar, and Srikanth 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.

[2] AMD Corporation. Leading the industry: Multi-core technology & dual-core processors from amd. `http://multicore.amd.com/en/Technology/`.

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

[4] C.B. Colohan, A. Ailamaki, J.G. Steffan, and T.C. Mowry. Optimistic Intra-Transaction Parallelism on Chip Multiprocessors. In *Proceedings of the 31st VLDB*, August 2005.

[5] Adrian Cristal, Daniel Ortega, Josep Llosa, and Mateo Valero. Out-of-order commit processors. In *Proceedings of the 10th HPCA*, February 2004.

[6] M.J. Garzarán, M. Prvulovic, J.M. 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.

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

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

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

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

[11] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of the 8th ASPLOS*, October 1998.

[12] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with transactional coherence and consistency (tcc). In *Proceedings of the 11th ASPLOS*, October 2004.

[13] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31th ISCA*, June 2004.

[14] Intel Corporation. Intel's dual-core processor for desktop PCs. http://www.intel.com/personal/desktopcomputer/dual_core/index.htm.

[15] N.P. 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.

[16] J. Kahle. Power4: A Dual-CPU Processor Chip. *Microprocessor Forum '99*, October 1999.

[17] V. Krishnan and J. Torrellas. The Need for Fast Communication in Hardware-Based Speculative Chip Multiprocessors. In *Proceedings of PACT '99*, October 1999.

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

[19] Jose Martínez, Jose Renau, Michael C. Huang, Milos Prvulovic, and Josep 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.

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

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

[22] 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.

[23] J. Oplinger, D. Heine, and M.S. Lam. In Search of Speculative Thread-Level Parallelism. In *Proceedings of PACT '99*, October 1999.

[24] M.K. 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.

[25] Milos Prvulovic, Maria Jesus Garzarán, Lawrence Rauchwerger, and Josep Torrellas. Removing architectural bottlenecks to the scalability of speculative parallelization. In *Proceedings of the 28th ISCA*, June 2001.

[26] G.S. Sohi, S. Breach, and T.N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd ISCA*, June 1995.

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

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

[29] J.G. Steffan, C.B. Colohan, and T.C. Mowry. Architectural Support for Thread-Level Data Speculation. Technical Report CMU-CS-97-188, School of Computer Science, Carnegie Mellon University, November 1997.

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

[31] J.G. Steffan, C.B. Colohan, A.B. Zhai, and T.C. Mowry. Improving Value Communication for Thread-Level Speculation. In *Proceedings of the 8th HPCA*, February 2002.

[32] J.G. Steffan and T.C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallellization. In *Proceedings of the 4th HPCA*, February 1998.

[33] Sun Corporation. Throughput computing—niagara. `http://www.sun.com/processors/throughput/`.

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

[35] Nathan Tuck and Dean M. Tullsen. Multithreaded value prediction. In *Proceedings of the 11th HPCA*, February 2005.

[36] T.N. Vijaykumar. Compiling for the multiscalar architecture. In *Ph.D. Thesis*, January 1998.

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

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

[39] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors. In *Proceedings of the 5th HPCA*, pages 135–141, January 1999.

A 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);
}
```

B 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.

B.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();
    }
   }
  }
}
```

B.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;
   }
  }
}
```

B.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();
    }
   }
  }
}
```

B.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();
    }
   }
  }
}
```

B.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();
    }
   }
  }
}
```