CMP Support for Large and Dependent Speculative Threads

Christopher B. Colohan, Anastasia Ailamaki, *Member*, *IEEE Computer Society*,
J. Gregory Steffan, *Member*, *IEEE*, and Todd C. Mowry

Abstract—Thread-level speculation (TLS) has proven to be a promising method of extracting parallelism from both integer and scientific workloads, targeting speculative threads that range in size from hundreds to several thousand dynamic instructions and which have minimal dependences between them. However, recent work has shown that TLS can offer compelling performance improvements when targeting much larger speculative threads of more than 50,000 dynamic instructions per thread with many frequent data dependences between them. To support such large and dependent speculative threads, the hardware must be able to buffer the additional speculative state and must also address the more challenging problem of tolerating the resulting cross-thread data dependences. In this paper, we present a chip-multiprocessor (CMP) support for large speculative threads that integrates several previous proposals for the TLS hardware. We also present a support for *subthreads*: a mechanism for tolerating cross-thread data dependences by checkpointing speculative execution. Through an evaluation that exploits the proposed hardware support in the database domain, we find that the transaction response time for three of the five transactions from TPC-C (on a simulated four-processor chip-multiprocessor) speed up by a factor of 1.9 to 2.9.

Index Terms—Multiprocessor systems, thread-level speculation, databases, cache coherence.

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, an increasingly important research question is how to ease the task of taking full advantage of these computational resources. One potential answer is *Thread-Level Speculation* (TLS) [1], [2], [3], [4], [5], which enables either a compiler [6], [7] or the programmer [8], [9] to optimistically and incrementally introduce parallelism into a program while always maintaining functional correctness.

Although 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 [1], [2], [3], [4], [5], a common theme in nearly all of the TLS studies to date is that they have focused on benchmark programs (for example, from the SPEC suite [10]), where the TLS threads are of modest size, typically a few hundred to a few thousand dynamic instructions per thread [6], [9], [11], or on benchmarks with very infrequent data dependences [12]. Despite these seemingly favorable conditions, TLS has had limited success on SPEC applications, offering the greatest benefits on scientific and CAD applications. In addition, most of the mechanisms for supporting TLS that have been proposed to date take an *all-or-nothing* approach [1], [2], [3], [4], [5] in the sense that a given thread only succeeds in improving overall performance if it suffers *zero* interthread dependence violations. In the case of a single violated dependence, the speculative thread is restarted at the beginning. Although such an all-or-nothing approach may make sense for the modest-sized and mostly independent threads typically chosen from SPEC benchmarks, this paper explores a far more challenging (and possibly more realistic) scenario: how to effectively support TLS when the thread sizes are much larger and where the complexity of the threads causes interthread dependence violations to occur far more frequently. With large threads and frequent violations, the all-or-nothing approach will result in more modest performance improvements at best.

In a recent paper [8], we demonstrated that TLS can be successfully used to parallelize *individual* transactions in a database system, improving transaction *latency*. This work is important to database researchers for two reasons: 1) Some transactions are latency sensitive, such as financial transactions in stock markets, and 2) reducing the latency of transactions that hold heavily contended locks allows the transactions to commit faster (and, hence, release their locks faster). Releasing locks more quickly reduces lock contention, which improves transaction throughput [13]. In this paper, we show that TLS support facilitates a nearly *twofold speedup* on a simulated four-way chip multiprocessor (CMP) for NEW ORDER, the transaction that accounts for almost half of the TPC-C workload [14], as illustrated later in Section 6.

- *C.B. Colohan is with Google Inc., 1600 Ampitheatre Parkway, Mountain View, CA 94043. E-mail: colohan@google.com.*
- *A. Ailamaki and T.C. Mowry are with the Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213. E-mail: {natassa, tcm}@cs.cmu.edu.*
- *J.G. Steffan is with the Department of Electrical and Computer Engineering, Computer Engineering Research Group, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada M5S 3G4. E-mail: steffan@eecg.toronto.edu.*

Manuscript received 25 Aug. 2006; revised 27 Jan. 2007; accepted 23 Feb. 2007; published online 13 Mar. 2007.
Recommended for acceptance by R. Iyer and D.M. Tullsen.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDSSI-0246-0806.
Digital Object Identifier no. 10.1109/TPDS.2007.1081.

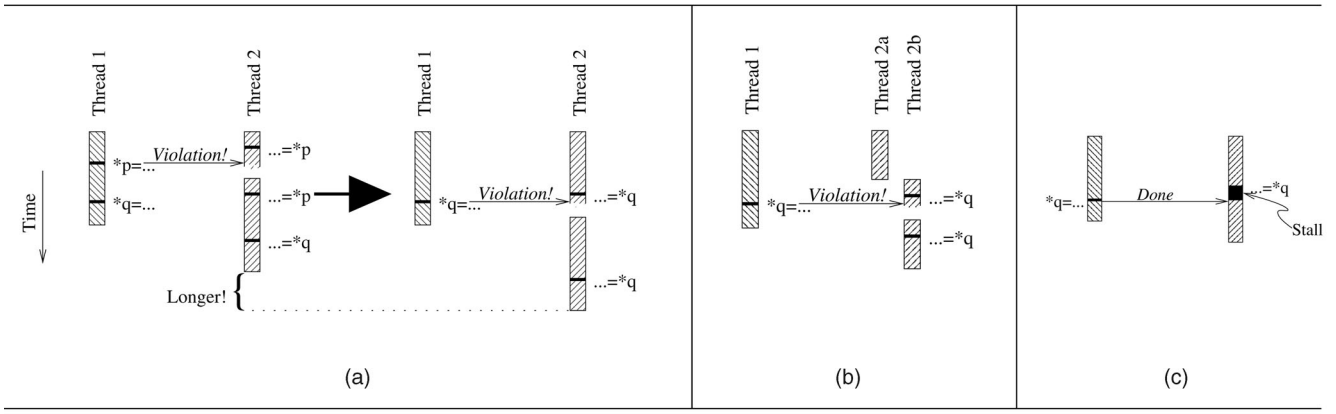


Fig. 1. The use of subthreads enables a performance tuning process, where eliminating data dependences improves performance. (a) Eliminating the dependence through *p can potentially *hurt* performance with an all-or-nothing TLS execution. (b) Execution with subthreads. (c) Idealized parallel execution.

To achieve such an impressive speedup, we first needed to overcome new challenges that did not arise for smaller programs such as the SPEC benchmarks. In particular, after breaking up the TPC-C transactions to exploit the most promising sources of parallelism in the transaction code, the resulting speculative threads 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 thread. Furthermore, there were far more interthread data dependences than in previous TLS studies. These dependences are due to internal database structures (that is, not the SQL code). As a result, we observed no speedup when we ran them on an existing all-or-nothing TLS architecture.

1.1 Supporting Large and Dependent Speculative Threads

For the TLS execution to speed up our speculatively parallelized database transactions, we have to overcome three major challenges. First, the baseline TLS hardware needs to be able to buffer all of the state generated by large speculative threads, as well as aggressively forward values between speculative threads to reduce dependence violations. Second, to enable incremental parallelization, 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. In this paper, we present a unified design incorporating several hardware techniques to address these two challenges, although, for some techniques, similar ones have been demonstrated previously in isolation.

The third challenge is the most crucial. With all-or-nothing TLS support, a dependence violation causes the entire speculative thread to restart instead of just reexecuting the instructions that misspeculated. Instead, we must be able to tolerate frequent unpredictable dependences between speculative threads and continue to extract partial parallelism in their presence. In this paper, we propose a mechanism for *tolerating* failed speculation by using lightweight checkpoints called *subthreads* to allow us to roll a failed thread back to an intermediate point before speculation failed.

Support for subthreads makes TLS more useful since it allows the programmer to treat the parallelization of a program as a *performance tuning* process. Without subthreads, TLS only provides a performance gain if a chosen thread decomposition results in speculative threads with infrequent data dependences between them. With subthreads, the programmer can engage in an iterative process:

1. decompose the program into speculative threads,
2. profile execution and observe which data dependence causes the most execution to be rewound,
3. modify the program to avoid this data dependence, and
4. iterate on this process (by returning to Step 2).

Without subthread support, removing a data dependence can actually *degrade performance*: As shown in Fig. 1a, removing the early dependence (through *p) only delays the inevitable reexecution of the entire speculative thread because of the later dependence (through *q). With subthread support—and an appropriate choice of subthread boundaries—each removed dependence will gradually improve performance (Fig. 1b). With enough subthreads per thread, TLS execution approximates an idealized parallel execution (Fig. 1c) where parallelism is limited only by data dependences, effectively stalling each dependent load until the correct value is produced. In summary, subthreads allow TLS to improve performance even when speculative threads have unpredictable and frequent data dependences between them.

1.2 Related Work

To implement both our shared cache TLS support and our subthread support, we store multiple versions of values in the cache. We essentially propose centralized hardware support for cache line versioning which is similar in spirit to the original *Address Resolution Buffer* (ARB) of the Multiscalar Architecture [15]—however, in our approach, we store speculative versions of lines in multiple levels of the cache hierarchy, allowing us to take advantage of the larger sizes and associativities of the caches further from the CPUs and to avoid increasing the complexity and hit latency of the first-level caches. Our experience with large and dependent speculative threads is that simple schemes without versioning and second-level storage are insuffi-

cient. Approaches proposed since the ARB such as the Speculative Versioning Cache (SVC) and the IACOMA [1], [16] approach also support speculative versions in a more generic cache hierarchy. By limiting the visibility of replicas to just the CPUs sharing an L2 cache, we avoid the need for the version ordering list used in SVC.

In this paper, we detail a design for a speculative victim cache, which we find to be sufficient to capture cache set overflows for our database applications. Two prior approaches address the problem of cache capacity overflow: Prvulovic et al. proposed a technique that allows speculative state to overflow into the main memory [16], and Cintra et al. proposed a hierarchical TLS implementation that allows the oldest thread in a CMP to buffer speculative state in the L2 cache (while requiring that the younger threads running on a CMP be restricted to the L1 cache) [17]. In this paper, we propose a similar hierarchical implementation with one important difference from Cintra et al.'s scheme: it allows *all* threads to store their speculative state in the larger L2 caches. With this support, 1) all threads can take advantage of the large size of the L2 cache, 2) threads can aggressively propagate updates to other more recent threads, and 3) we can more easily implement *subthreads*, described later in Section 3.

Subthreads are a form of checkpointing and, in this paper, we use such checkpointing to tolerate failed speculation. Prior work has used checkpointing to simulate an enlarged reorder buffer with multiple checkpoints in the load/store queue [18], [19] and a single checkpoint in the cache [20]. Martínez et al.'s checkpointing scheme [20] effectively enlarges the reorder buffer and is also integrated with TLS and is thus the most closely related work. The subthread design we present in this paper could be used to provide a superset of the features in Martínez et al.'s work at a higher cost: subthreads 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 simultaneous multithreading (SMT) processor could be used to checkpoint the system and recover from failed value prediction, expanding the effective instruction window size [21]—the techniques we use to create subthreads could also be used to create checkpoints at high-confidence prediction points following Tuck and Tullsen's method.

For TLS, tolerating failed speculation using subthreads implies tolerating data dependences between speculative threads. A recent alternative approach for tolerating cross-thread dependences is *selective reexecution* [22], where only the slice of instructions involved in a violated dependence is reexecuted; although we do not evaluate this here, we may be able to use subthreads in conjunction with selective reexecution to help tolerate potentially unsafe slices. Other studies have explored predicting and learning data dependences and turning them into synchronization [23], [24], [11] or have used the compiler to mark likely dependent loads and tried to predict the consumed values at runtime [11]. Initially, we tried to use an aggressive dependence predictor like the one proposed by Moshovos et al. [24], but found that only one of the several dynamic 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. Support for subthreads provides a more elegant solution that is *complementary* to hardware-based prediction and software-based synchronization techniques since using subthreads significantly reduces the high cost of misspeculation.

The notion of using speculative execution to simplify manual parallelization was first proposed by Prabhu and Olukotun for parallelizing SPEC benchmarks on the Hydra multiprocessor [2], [9]. 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 nonspeculative work in a single thread, as well as allowing partial rollback of a thread through subthreads. Hammond et al. push the idea of programming with threads to its logical extreme such that the program consists of nothing but threads—resulting in a vastly simpler architecture [25], but requiring changes to the programming model to accommodate that new architecture [26]. The architecture presented in this paper is closer to existing CMP architectures and can execute both regular binaries and those that have been modified to take advantage of TLS.

2 Two-Level Speculative State Tracking and Buffering

TLS allows us to break a program's sequential execution into parallel speculative threads and ensures that *data dependences* between the newly created threads are preserved. Any read-after-write dependence between threads that is *violated* must be *detected* and *corrected* by restarting the offending thread. Hardware support for TLS makes the detection of violations and the restarting of threads inexpensive [2], [3], [16], [27].

Our database benchmarks stress TLS hardware support in new ways that have not been previously studied since the threads are necessarily so much larger. In previous work, speculative threads of various size ranges were studied including 3.9-957.8 dynamic instructions [6], 140-7,735 dynamic instructions [9], 30.8-2,252.7 dynamic instructions [11], and up to 3,900-103,300 dynamic instructions [12]. The threads studied in this paper are quite large, with 7,574-489,877 dynamic instructions. These larger threads present two challenges. First, more speculative state has to be stored for each thread: from 3 Kbytes to 35 Kbytes; the additional state is required to store multiple versions of cache lines for subthreads. Most existing approaches to TLS hardware support cannot buffer this large amount of speculative state. Second, these larger threads have many data dependences between them that cannot be easily synchronized and forwarded by the compiler [7], 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 threads attractive.

In this section, we describe the underlying CMP architecture and how we extend it to handle large speculative threads. We assume a CMP where each core has a private L1 cache and multiple cores share a single chip-wide L2 cache. For simplicity, in this paper, each CPU executes a single speculative thread. We buffer speculative state in the caches, detecting violations at a cache line granularity. Both the L1 and L2 caches maintain speculative

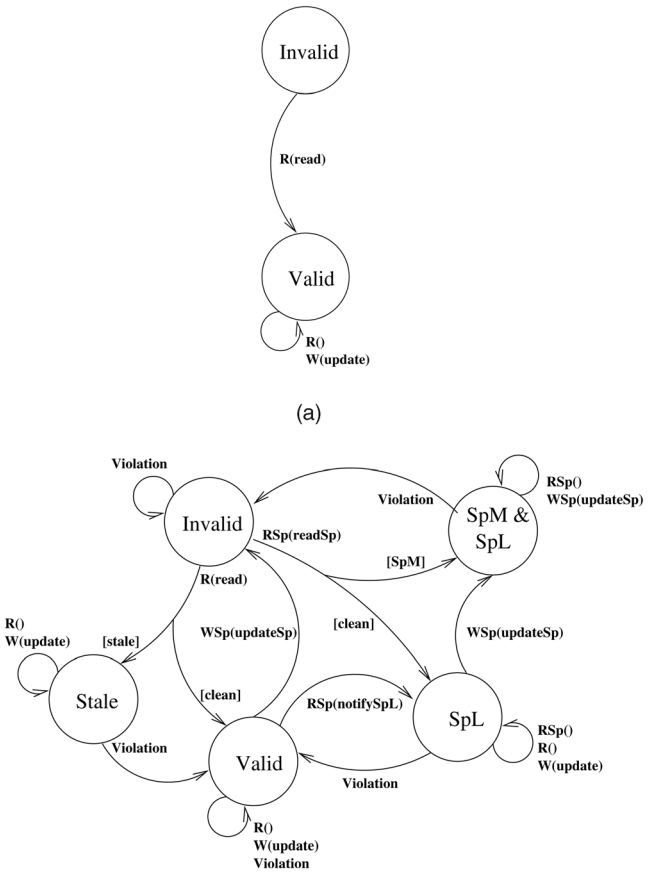


Fig. 2. L1 cache line state transition diagram. Any transitions not shown (for example, action **R** for a line in the state ***SpM***) is due to an impossible action: The tag match will fail and the transition will be treated as a miss. Line states are explained in Table 1. Edges are labeled with the processor action (Table 1), followed by the message to the L2 cache in round brackets (Table 1). Square brackets show responses from the L2 cache (Table 1). (a) Basic write-through protocol. (b) Enhanced with TLS support.

state: Each L1 cache buffers cache lines that have been speculatively read or modified by the thread executing on the corresponding CPU, whereas the L2 caches maintain inclusion and buffer copies of all speculative cache lines that are cached in any L1 on that same chip. The detection of violations between speculative threads running on *the same chip* is performed within the L2 cache through the two-level protocol described below. For the detection of violations between speculative threads running on *different chips*, we assume support for an existing scheme for distributed TLS cache coherence, such as the STAMPede protocol [3].[1]

2.1 L1 Cache State

Our design has the following goals: 1) to reduce the number of dependence violations by aggressively propagating store values between threads and 2) 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 speculative threads.

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.

TABLE 1
Protocol States, Actions, and Messages

Cache line states

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

Actions

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

Messages from L1 to L2 cache

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

Responses from L2 cache to L1 cache

clean	Requested line *has not* been speculatively modified by any earlier thread.
stale	Requested line *has* been speculatively modified by the current or an earlier thread, but returning non-speculative (committed) version of the line since speculation is currently escaped.
SpM	Requested line *has* been speculatively modified by an earlier thread, returning speculative version of the line.

In Fig. 2, we illustrate how to take a simple write-through L1 cache line state transition diagram and enhance it for use with our shared L2 cache design. In the following discussion, we explain the new states and the functionality they offer (Table 1).

To track the first speculative load of a thread, 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 that 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 that triggers it cannot complete until the cache miss is serviced. The *notifySpL* message is nonblocking, so the load that 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 that would result in missing a violation, all *notifySpL* messages must be processed by the L2 cache before a speculative thread 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 a thread. This effectively makes each speculative thread 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 cache lines (if the L2 cache receives a *notifySpL* message for a line that has been speculatively modified by an earlier thread, then this will generate a violation for the loading thread). In Section 6.3, we evaluate the performance impact of this optimization.

Detecting the *last* store to a memory location by a speculative thread is more challenging. Although we do not evaluate it here, a sophisticated design could combine a write-back cache with a *last-touch predictor* [28] to notify the L2 of only the *last* store performed by a thread. 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 threads may load those values to avoid dependence violations. Each L1 cache line also has a *speculatively modified (SpM) bit*, which is set on the first speculative store so that it can be flash-invalidated if the thread violates a dependence (rather than relying on an invalidation from the L2 cache).

In our design, when the L1 cache holds a line that is marked speculative, we assume that it is up to date (that is, it contains all changes made by older speculative threads). Since the merging of writes from different speculative threads is performed in the L2 cache, the L1 cache cannot transition a *Valid* line into an *SpM* line without querying the L2 cache. Because of this, in Fig. 2, you will see that a speculative write to a *Valid* line invalidates the line so that any loads to that line retrieve the correct line from the L2 cache.

When a speculative thread commits, the L1 cache can simply clear all of the *SpM* and *SpL* bits since none of the state associated with the committing thread or any earlier thread is speculative. If multiple threads share a single L1 cache, then the *SpM* and *SpL* bits can be replicated, one per thread, as is done for the shared cache TLS protocol proposed in prior work [29].

The *Stale* state in Fig. 2 is used for temporarily escaping speculation and is discussed further in Section 4.2.

2.2 L2 Cache State

The L2 cache buffers speculative state and tracks data dependences between threads using the same techniques as the TLS protocol proposed in prior work [29]. 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 a speculative thread number and these numbers are mapped onto the *speculative thread contexts* in the L2 cache. Each speculative thread context represents the state of a running speculative thread. Each cache line has an *SpM* bit and *SpL* bit per speculative thread context. The dependences between threads sharing the same L2 cache are detected when an update is processed by checking for *SpL* bits set by later threads. The dependences between threads running on different L2 caches are tracked using the extended cache coherence proposed in prior work [29].

Given the large speculative threads that are found in database transactions, we need to be able to store large amounts of speculative state in the L2 cache from multiple speculative threads. Often, there are two *conflicting* versions of a line that need to be stored—for example, if a line is modified by different threads, then each modification must be tracked separately so that violations can efficiently undo the work from a later thread without undoing the work of an earlier thread. 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 previously studied in the Multiscalar project [1]; we present our simpler replication scheme in detail in Section 2.3.

If a speculative line is evicted from the cache, then we need to continue tracking the line's speculative state. With large speculative threads and cache line replication, this becomes more of a problem than it was in the past: We use a *speculative victim cache* [30], [31] to hold the evicted speculative lines and track violations caused by these lines. We have found that a small victim cache is sufficient to capture the overflow state, but, if more space is required, we could use a memory-based overflow area such as that proposed by Prvulovic et al. [16].

2.3 Cache Line Replication

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

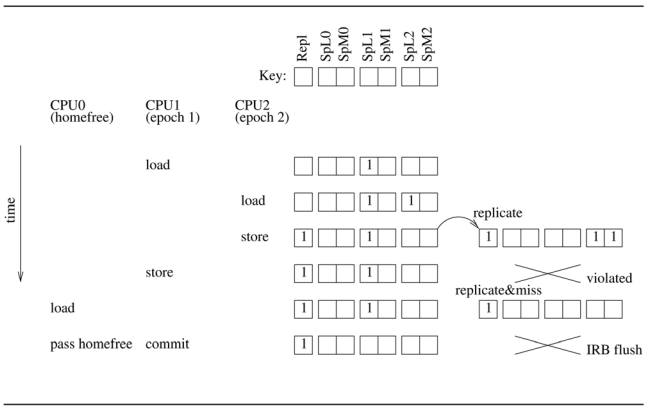


Fig. 3. Step-by-step example demonstrating cache line replication for a single cache line. Cache line replication avoids violations when a *replica conflict* occurs.

What can we do if a replica conflict arises? Every replica conflict involves two speculative threads: If the replica conflict arises when a store is performed by the later thread, then the later thread can be stalled until the earlier thread commits; if the replica conflict arises when a load or store is performed by the earlier thread, then it can be resolved by violating the later thread. Alternative approaches are to stall a thread until it is the oldest thread (that is, it will definitely commit) when a replica conflict is detected and to violate a thread when a replica conflict is detected. Both of these approaches hurt performance severely if replica conflicts occur 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 (Fig. 3). As speculative threads commit, these replicas are combined into a single line.

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

Because the most recent replica with respect to the current thread must be located, the 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 that indicates that a replica of the line *may* exist, which is set whenever a new replica 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 speculative threads that are older than the current thread are cleared on the newly created line, whereas the bits representing the current thread and newer threads are cleared on the source line. This way, the responsibility for tracking speculative state is divided so that the older state resides on the source line and the newly created replica tracks the state associated with the current and later threads.

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 a thread commits, all speculatively modified cache lines associated with the committing thread 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 and we ensure this by having the commit operation first invalidate any clean replicas that are made obsolete by the commit operation. The lines that need to be invalidated are tracked through the *invalidation required buffer* (IRB), which operates in a similar fashion to the ownership required buffer (ORB) [29]. There is one IRB per speculative thread context. An IRB entry is simply a cache tag that says "the cache line associated with this address and thread may have an older replica." When a replica is created, it may generate up to two IRB entries. If there exists any replica with a state older than the newly created replica's state, then an entry is added to the newly created replica's IRB. If there exist any replicas with a 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 a thread is violated, the IRB for that thread is cleared. When a thread commits, the cache first invalidates any clean replicas named by the IRB entries and then clears the *SpM* and *SpL* bits associated with the committing thread. We found that an IRB of 32 entries is sufficient to capture the majority of invalidation-required information; should the IRB overflow, correctness is preserved by violating the logically latest thread.

The L2 cache maintains inclusion and, also, directory bits that track which L1 caches may have a replica of a given line so that, when a replica is updated or invalidated, an invalidation is sent to the appropriate L1 caches.

To briefly summarize the impact of our support, we have added two bits per speculative context plus one bit per line to indicate whether a replica exists. In our most aggressive implementation, this results in 17 bits added per L2 cache line. The L2 cache lookup latency may be slower when a replica exists, since we now have to find the appropriate replica in either the corresponding cache set or the victim cache.

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

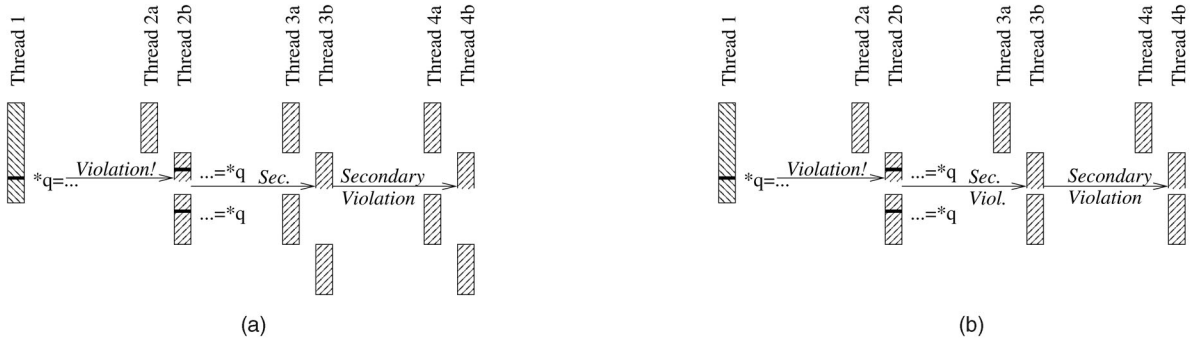


Fig. 4. The effect of secondary violations with and without subthread dependence tracking. (a) All later subthreads restart. (b) Only dependent subthreads restart.

In summary, this design supports the efficient buffering of speculative state and dependence tracking for large speculative threads, giving them access to the full capacity of the L1 and L2 caches. Store values are aggressively propagated between threads, reducing violations. A more detailed description of our underlying hardware support is available in a technical report [32].

3 Tolerating Dependences with Subthreads

The motivation for subthreads is to tolerate data dependences between speculative threads. Previously proposed hardware mechanisms that also tolerate data dependences between speculative threads include *data dependence predictors* and *value predictors* [24], [11]. A data dependence predictor automatically synchronizes a dependent store/load pair to avoid a dependence violation, whereas a value predictor provides a predicted value to the load, which can then proceed independently from the corresponding store. In our experience with database benchmarks, the resulting large speculative threads contain between 20 and 75 dependent loads per thread *after* iterative optimization and we found that previous techniques for tolerating those dependences were ineffective. However, we note that subthreads are *complimentary* to prediction since the use of subthreads reduces the penalty of any misprediction.

Subthreads are implemented by extending the L2 cache such that, for every single speculative thread, the L2 cache can maintain speculative state for multiple thread contexts. For example, if we want to support two subthreads for each of four speculative threads (eight subthreads total), then the L2 cache must be extended to maintain speculative state for eight distinct thread contexts. At the start of a new subthread, the register file is backed up and all subsequent speculative state is saved in the next thread context. Although we do not model a particular implementation of a register file back-up, this could be accomplished quickly through shadow register files or more slowly by backing up to the memory. New subthreads can be created until all of the thread contexts allocated to the corresponding speculative thread have been consumed.

In our TLS hardware support, dependence violations are detected between thread contexts; when subthreads are supported, any violation will specify both the thread *and subthread* that needs to be restarted. Within a speculative thread, the subthreads execute serially and in-order; hence, there are no dependence violations between them. Note that, in our design so far, the L1 caches are unaware of subthreads: Dependence tracking between subthreads is performed at the L2 cache. Any dependence violation results in the invalidation of all speculatively modified cache lines in the appropriate L1 cache and any necessary state can be retrieved from the L2 cache. To reduce these L1 cache misses on a violation, the L1 cache could also be extended to track subthreads; however, we have found this support to be not worthwhile. In summary, no additional hardware is required to detect dependence violations between threads at a subthread granularity other than providing the additional thread contexts in the L2 cache.

When a speculative thread violates a data dependence, we call this a *primary violation*. Since logically, later speculative threads may have consumed the incorrect values generated by the primary violated thread, all of these later threads must also be restarted through a *secondary violation*. With support for subthreads, we can do better, as shown in Fig. 4. In Fig. 4a, when the dependence violation between thread 1 and subthread $2b$ is detected, all of thread 3's and 4's subthreads are restarted through secondary violations, even though subthreads $3a$ and $4a$ completed before subthread $2b$ started. Since subthreads $3a$ and $4a$ could *not* have consumed any data from $2b$, it is unnecessary to restart them. Hence, we prefer the execution shown in Fig. 4b, which requires that we track the temporal relationship between subthreads by having every speculative thread maintain a *subthread start table*, which is described next.

3.1 Subthread Start Table

To implement the desirable recovery model shown in Fig. 4b, we need to track when subthreads start so that we know the relative starting points of subthreads across different speculative threads—this is supported using the *subthread start table* and a simple numbering of subthreads within each speculative thread. When a speculative thread suffers a dependence violation, it notifies all logically, later speculative threads that they must also restart (that is, that they have suffered a secondary violation) and indicates its own thread ID and the subthread number it is currently executing. When a logically-later speculative thread receives such a notification of a secondary violation, it uses the received thread ID and subthread number to look up in the table which of its own

subthreads were executing when the indicated subthread began execution and then rewinds its own execution to the start of that subthread. To maintain this information, the subthread start table must be notified whenever a speculative thread starts, a subthread starts, or a violation occurs.

Later, in Section 6.2, we investigate the trade-offs in the frequency of starting new subthreads and the total number of subthreads supported. Next, we look at how our support for large speculative threads and subthreads leads to an elegant model for iterative feedback-driven parallelization.

4 Exploiting Subthreads: Tuning Parallel Execution

We have applied TLS to the loops of the transactions in TPC-C and found that the resulting threads are large (7,574-489,877 average dynamic instructions) and contain many interthread data dependences (an average of 292 dependent dynamic loads per thread for the NEW ORDER transaction), which form a critical performance bottleneck. Our goal is to allow the programmer to treat parallelization of transactions as a form of *performance tuning*: A profile guides the programmer to the performance hot spots, and performance is improved by modifying only the critical regions of code. To support this iterative parallelization process, we require the hardware to provide a profile feedback reporting which store/load pairs triggered the most harmful violations—those that caused the largest amount of failed speculation.

4.1 Profiling Violated Interthread Dependences

Hardware support for profiling interthread dependences requires only a few extensions to the basic TLS hardware support. Each processor must maintain an *exposed load table* [11]—a moderate-sized direct-mapped table of PCs, indexed by a cache tag, which is updated with the PC of every speculative load that is *exposed* (that is, has not been preceded in the current subthread by a store to the same location—as already tracked by the basic TLS support). Each processor also maintains cycle counters that measure the duration of each subthread. When the L2 dependence tracking mechanism observes that a store has caused a violation, 1) the store PC is requested from the processor that issued the store and 2) 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 subthread 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 to provide the programmer with a profile of the problem load/store pairs, who can use the cycle counts to order them by importance.

4.2 Hardware Support for Escaping Speculation

When parallelizing speculatively within database transactions, there are often operations such as `malloc` that cause unnecessary dependences between speculative threads: The dependences are unnecessary because the order of these operations across speculative threads is unimportant to the semantics of the transaction. If there was a way to execute such operations nonspeculatively, then any dependences between them could be tolerated. Such an optimization also requires that, in the case where the speculative thread fails, the operation must be undone (for example, `free`). We call operations for which this optimization is possible *isolated undoable operations* (IUOs), which are described in further detail in a previous publication [8].

To optimize IUOs, the TLS hardware must support a method for *escaping speculation* by temporarily treating the executing thread as *nonspeculative*. This means that any loads by the escaped thread return committed to the memory state and not the speculative memory state. Any stores performed by the escaped thread are not buffered by the TLS mechanism and are immediately visible to all other threads. A side effect of this is that if an escaped thread writes to a location speculatively loaded by any speculative thread, *including itself*, it can cause that thread to be violated. The only communication between the speculative execution preceding the escaped speculation and the escaped thread is through registers, which can be carefully checked for invalid values caused by misspeculation. To avoid false dependences caused by writing temporary variables and return addresses to the stack, the escaped thread should use a separate stack (using a mechanism such as *stacklets* [33], [34]).[3] If an escaped thread 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 thread loads data into the L1 cache, it may perform loads that evict lines that were speculatively modified by the current thread. If the thread resumes the speculative execution and then loads the same line, it must receive the speculative evicted copy of the line and not the clean copy that 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 versions of a cache line (an alternative solution that should have a similar performance is to put a victim cache underneath the L1 cache to catch these ping-ponging lines).

Since the escaped code takes nonspeculative actions, the wrapper around it has to be carefully written to avoid causing harm to the rest of the program when misspecula-

3. TLS already assumes that each executing speculative thread has a private *stacklet* for storing local variables and the return addresses of called functions. If two threads shared a common stack, then an action as simple as calling a function would cause a violation since each thread 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 speculative threads (the compiler can check this assumption at compile time) and give each speculative thread 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 thread when it escapes speculation.

tion happens. For example, a misspeculating thread may go into an infinite loop allocating memory. The misspeculating thread must not be allowed to allocate so much memory that allocation fails for the least speculative thread. This potential problem can be avoided through software: One way is to place limits on the amount of memory allocated by a speculative thread. Another solution is to have the least speculative thread first violate any speculative threads (and, hence, have them release all resources) before allowing any allocation request to fail.

4.3 Tuning Parallelized Transactions

Previous work [8] has demonstrated an iterative process for removing performance-critical data dependences:

1. execute the speculatively parallelized transaction on a TLS system,
2. use profile feedback to identify the most performance-critical dependences,
3. modify the database management system (DBMS) code to avoid violations caused by those dependences, and
4. repeat.

Going through this process reduces the total number of data dependences between threads (from 292 dependent loads per thread to 75 dependent loads for NEW ORDER) and, more importantly, removes dependences from the critical path.

5 Experimental Framework

In this section and the next, we evaluate subthreads and our hardware support for large speculative threads in the context of speculatively parallelized database transactions.

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).[4] 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. We use BerkeleyDB since it is well written and is structured similarly to a modern database system back end (supporting features such as transactional execution, locking, a buffer cache, B-trees, and logging). 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 model a system with a memory-resident data set by configuring the DBMS with a large (100 Mbytes) buffer pool.[5] 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 benchmarks execute as follows:

1. start the DBMS,
2. execute 10 transactions to warm up the buffer pool,
3. start timing,
4. execute 100 transactions, and
5. stop timing.

All codes are compiled using `gcc` 2.95.3 with O3 optimization on an SGI MIPS-based machine. The BerkeleyDB database system is compiled as a shared library which is linked with the benchmarks that contain the transaction code. To apply TLS to each 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 [8], we described how we iteratively optimized the database system for TLS using the methodology described in Section 4. We evaluate the hardware using fully optimized benchmarks.

5.2 Simulation Infrastructure

We perform our evaluation using a detailed trace-driven simulation of a CMP composed of four-way-issue out-of-order superscalar processors similar to the MIPS R14000 [36], 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 that 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 and contention) are all modeled and are parameterized as shown in Table 2. For now, we model a zero-cycle latency for the creation of a subthread, including a register backup. This is a user-level simulation: All instructions in the transactions and DBMS are simulated, whereas the operating system code is omitted. 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 the main memory.

6 Experimental Results

6.1 Benchmark Characterization

We begin our evaluation by characterizing the benchmarks themselves, so we can better understand them. Table 3 shows some basic measurements of the benchmarks we study, including the number of processor cycles to execute a

4. Our workload was written to match the TPC-C spec as closely as possible, but it 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 benchmarks. 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.

5. This is roughly the size of the entire data set for a single warehouse. This ensures that reads will not go to a disk and data updates are performed at transaction commit time, consume less than 24 percent of execution time [35], and can be executed in parallel with other transactions.

TABLE 2
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	10 cycles
Minimum Miss Latency to Local Memory	75 cycles
Main Memory Bandwidth	1 access per 20 cycles

sequential version of each benchmark and the fraction of execution time of each benchmark that has been speculatively parallelized—which ranges from 30 percent to 99 percent. Examining the speculative threads for each benchmark, we see that they are indeed large: averaging from 8,000 instructions up to 490,000 instructions per speculative thread. Finally, the number of parallel speculative threads per instance of a transaction varies widely across the benchmarks, from only two speculative threads for PAYMENT to 191.7 speculative threads for STOCK LEVEL.

As a starting point for comparison, in Fig. 5, we run each 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 has four CPUs). This SEQUENTIAL experiment is normalized to 1.0 in Fig. 5—note that the large percentage of *Idle* is caused by three of the four CPUs idling in a sequential execution.

When we apply TLS without subthread support on a four-CPU system, shown in the NO SUBTHREADS bars, performance improves for the majority of the benchmarks. In the NEW ORDER and DELIVERY benchmarks, we observe that a significant fraction of the CPU cycles are spent idling: NEW ORDER does not have enough threads to keep four CPUs busy and, so, the NEW ORDER 150 benchmark is scaled to increase the number of threads by a factor of 10 and, hence, avoids this performance bottleneck. In the DELIVERY benchmark, we have parallelized an inner loop with only 63 percent coverage—in DELIVERY OUTER, the outer loop of the transaction is parallelized, which has 99 percent coverage, but increases the average thread size from 33,000 dynamic instructions to 490,000 dynamic instructions. With larger threads, the penalty for misspeculation is much higher and this shows up as a much larger

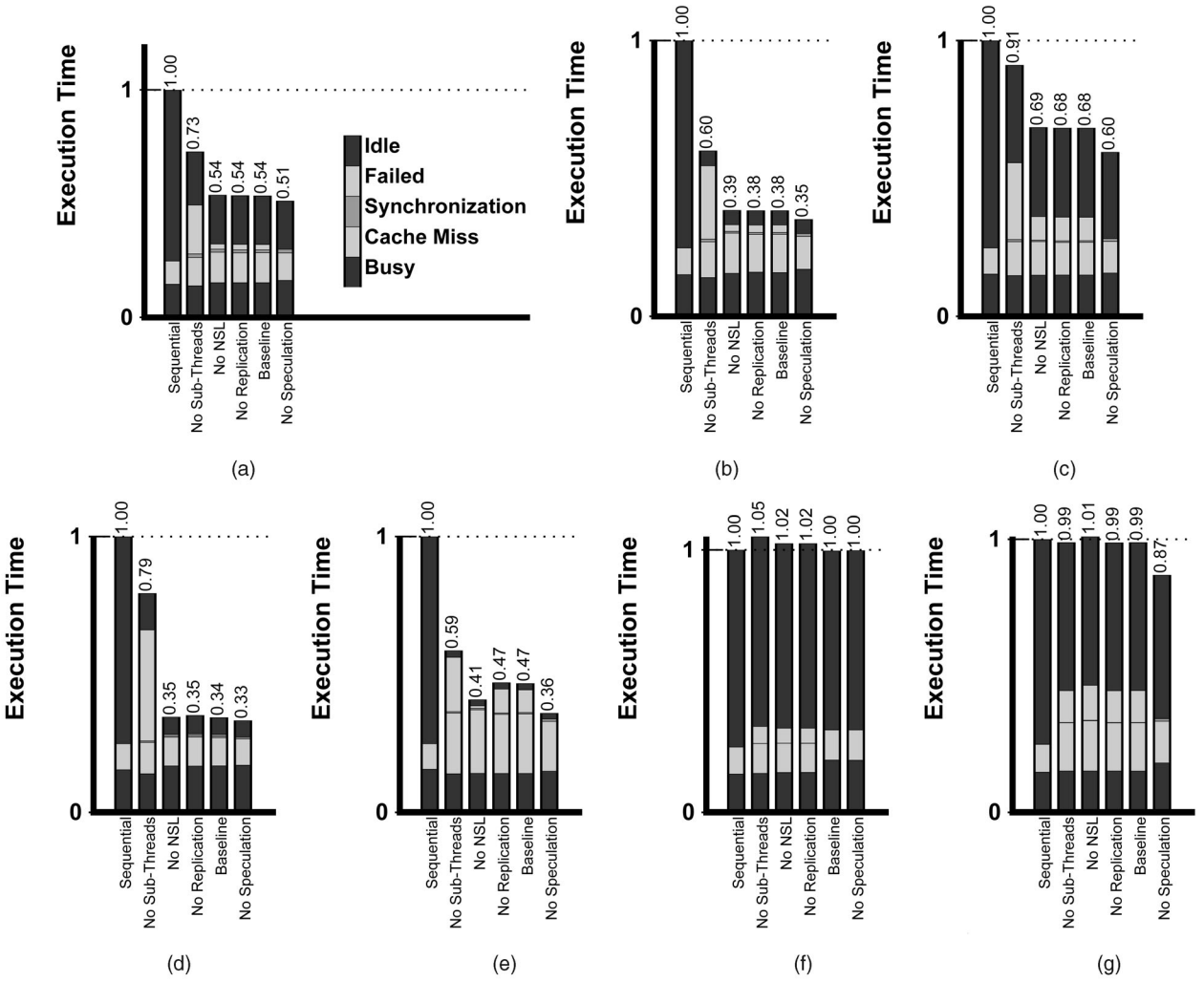
TABLE 3
Benchmark Statistics

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

Failed component in the NO SUBTHREAD bar of the DELIVERY OUTER graph—in fact, we observed that every speculative thread in every benchmark suffers a dependence violation at least once. Performance gain is still possible in this case since these violations can artificially synchronize execution until it is staggered but still enjoying a parallel overlap. The PAYMENT and ORDER STATUS transactions do not improve with TLS since they lack significant parallelism in the transaction code and, so, we omit them from further discussion. By parallelizing the execution over four CPUs, we also execute using four L1 caches. For the STOCK LEVEL transaction, this increases the time spent servicing cache misses significantly as it shares data between the L1 caches.

As an upper bound on performance, in the NO SPECULATION experiment, we execute purely in parallel—incorrectly treating all speculative memory accesses as nonspeculative and, hence, ignoring all data dependences between threads. This execution shows a sublinear speedup due to the nonparallelized portions of execution (Amdahl's law) and due to a loss of locality and communication costs due to the spreading of data across four caches [37]. This experiment shows an upper bound on the performance of TLS since it represents an execution with no dependence violations. From this, we learn that all of our benchmarks except for PAYMENT and ORDER STATUS can improve dramatically if the impact of data dependences is reduced or removed.

The BASELINE experiment in Fig. 5 shows a TLS execution with eight subthreads per speculative thread and 5,000 instructions per subthread. The subthread support is clearly beneficial, achieving a speedup of 1.9 to 2.9 for three of the five transactions. Subthreads reduce the time spent on failed speculation for both NEW ORDER variants, both DELIVERY variants, and STOCK LEVEL. The resulting execution time for both NEW ORDER variants and DELIVERY OUTER is very close to the NO SPECULATION execution time, implying that further optimizations to reduce the impact of data dependences are not likely to be worthwhile for these benchmarks. For DELIVERY, STOCK LEVEL, and ORDER STATUS, it appears in the graph that there is still room for improvement, but a hand analysis determined that the remaining time spent on failed speculation is due to actual data dependences that are difficult to optimize away in the code. The improvement due to subthreads is most dramatic when the speculative threads are largest: The DELIVERY OUTER benchmark executes more than twice as quickly with subthread support enabled than without. Subthreads do not have a large impact on the time spent servicing cache misses: This shows that the additional cache state required to support sub-



Bar	Explanation
Sequential	No modifications or TLS code added.
No Sub-Threads	Baseline execution with sub-thread support disabled.
No NSL	Baseline execution without the use of *notifySpL* messages from the L1 to L2 cache.
No Replication	Baseline execution without support for replication in the L1 cache.
Baseline	Execution on hardware described in this article.
No Speculation	Upper bound—modified hardware to treat all speculative writes as non-speculative.

Fig. 5. Performance of optimized benchmarks on four CPUs run sequentially, without subthread support, without the *notifySpL* message, without replication in the L1 cache, with full support, and with an upper bound where there is no speculation. (a) NEW ORDER. (b) NEW ORDER 150. (c) DELIVERY. (d) DELIVERY OUTER. (e) STOCK LEVEL. (f) PAYMENT. (g) ORDER STATUS.

threads does not exceed the capacity of the L2 cache. Note that we will return to discussing the *No NSL* and *No Replication* results later.

6.2 Choosing Subthread Boundaries

Subthreads allow us to limit the amount of execution rewound on a misspeculation, but how frequently should we start a new subthread and how many subthreads are necessary for good performance? We want to minimize the number of subthread contexts supported in the hardware and we also want to understand the performance benefits of increasing the number of contexts. Since interthread dependences are rooted at loads from the memory, we want to start new subthreads just before certain loads. This leads to two important questions. First, is a given load likely to cause a dependence violation? We want to start subthreads before loads that 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 [11]. Second, if the load does cause a violation, how much correct execution will the subthread avoid rewinding? If the load is near the start of the thread or a previous subthread, then a violation incurred at this point will have a minimal impact on performance. Instead, we would rather save the valuable subthread context for a more troublesome load. A simple strategy that works well in practice is to start a new subthread every nth speculative instruction for a careful choice of n.

In Fig. 6, we show an experiment where we varied the number of subthreads available to the hardware and varied the spacing between subthread start points. We would expect that the best performance would be obtained if the use of subthreads is conservative since this minimizes the number of replicate versions of each speculative cache line

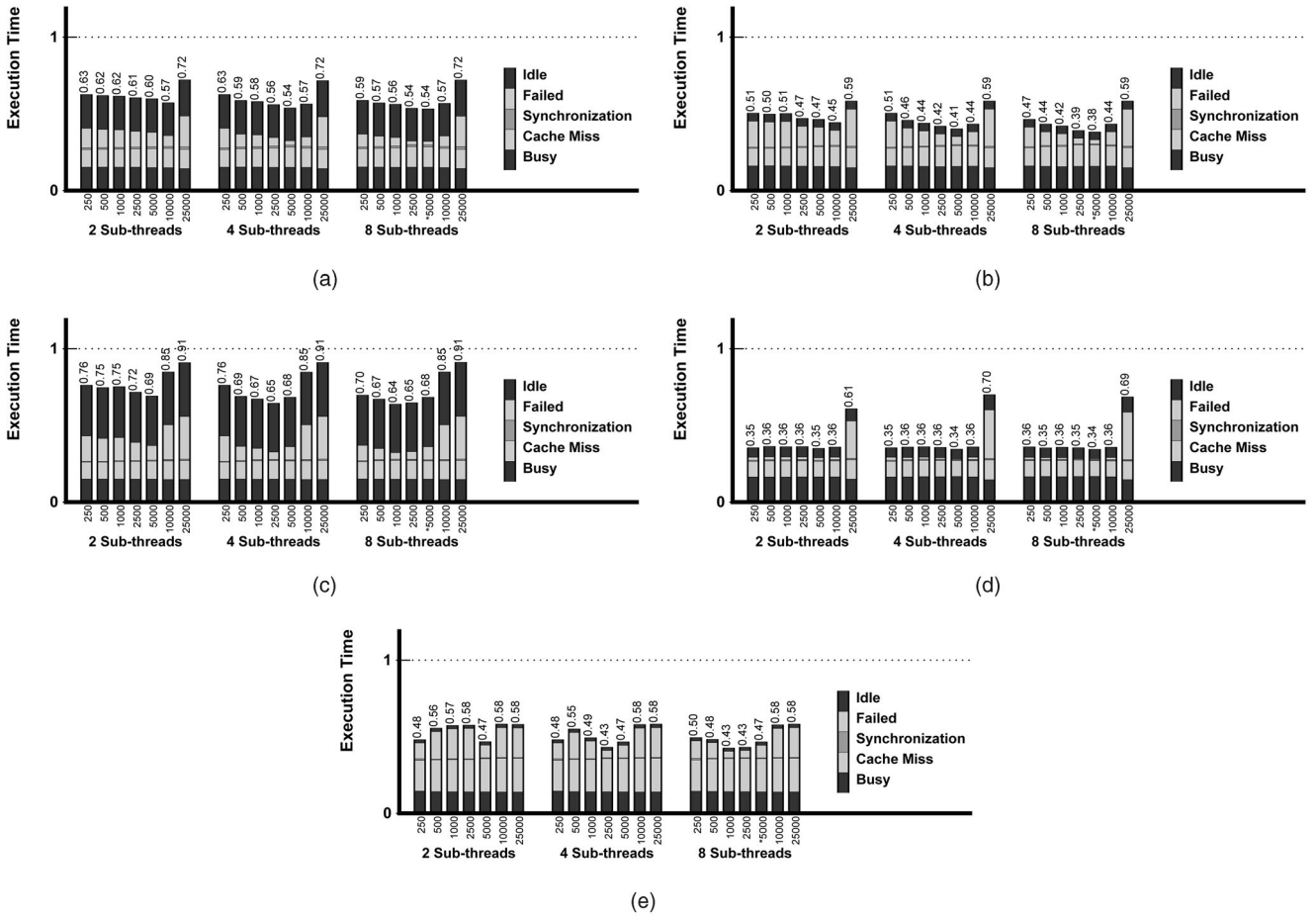


Fig. 6. Performance of optimized benchmarks on a four-CPU system when varying the number of supported subthreads per thread from two to eight, varying the number of speculative instructions per subthread from 250 to 25,000. The BASELINE experiment has eight subthreads and 5,000 speculative instructions per thread. (a) NEW ORDER. (b) NEW ORDER 150. (c) DELIVERY. (d) DELIVERY OUTER. (e) STOCK LEVEL.

and, hence, minimizes cache pressure. Since each subthread requires a hardware thread context, using a small number of subthreads also reduces the amount of required hardware. A subthread would ideally start just before the first misspeculating instruction in a thread so that, when a violation occurs, the machine rewinds no farther than required.

If hardware could predict the first dependence very accurately, then supporting two subthreads per thread would be sufficient. With two subthreads, the first subthread would start at the beginning of the thread 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 subthreads periodically as the thread executes.

In Fig. 6, we vary both the number and size of the subthreads used for executing each transaction. Surprisingly, adding more subthreads does not increase cache pressure enough to have a negative impact on performance—instead, the additional subthreads serve to either increase the fraction of the thread that is covered by subthreads (and, hence, protected from a large penalty if a violation occurs) or increase the density of subthread start points within the thread (decreasing the penalty of a violation).

When we initially chose a distance between subthreads of 5,000 dynamic instructions, it was somewhat arbitrary: We chose a round number that could cover most of the NEW ORDER transaction with eight subthreads per thread. This value has proven to work remarkably well for all of our transactions. A closer inspection of both the thread sizes listed in Table 3 and the graphs in Fig. 6 reveals that, instead of choosing a single fixed subthread size, a better strategy may be to customize the subthread size such that the average thread size for an application would be divided evenly into subthreads.

One interesting case is DELIVERY OUTER, in Fig. 6d, where a data dependence early in the thread's execution causes all but the nonspeculative thread to restart. With small subthreads, the restart modifies the timing of the thread's execution such that a later data dependence occurs in order, avoiding violations. Without subthreads or with very large subthreads (such as the 25,000 case in Fig. 6d), this secondary benefit of subthreads does not occur.

6.3 Sensitivity to L1 Cache Implementation

When implementing IUOs, we found that the L1 cache often suffered from thrashing, caused by a line that was repeatedly loaded by a speculative code and then loaded by a suspended code. To combat this effect, we added replication to the L1 cache, which lets an L1 cache hold both a speculative version and a nonspeculative version of a cache line simultaneously. In the NO REPL bar in Fig. 5, 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 nonspeculative 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 speculative loaded* (NSL) message. This design optimistically assumes that the nonspeculative 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 Fig. 5, we remove this message and cause a speculative load of a line that exists nonspeculatively in the L1 cache to be treated as an L1 cache miss. We see that the NSL message offers a very minor performance gain to NEW ORDER, NEW ORDER 150, DELIVERY, and DELIVERY OUTER, but the optimism causes a nontrivial slowdown for STOCK LEVEL. As a result, we conclude that, unless the optimism that the NSL message offers is tempered, the NSL bit is a bad idea.

6.4 Sensitivity to Victim Cache Implementation

Since we noticed that the cache line replication caused by subthreads exacerbates the problem of failed speculation caused by evicted speculative cache lines, we have added a speculative victim to our design to avoid such 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 four-way L2 cache and using a four-CPU machine, we found that only one application (NEW ORDER 150) uses all 64 of the entries in the victim cache. If we increase the associativity of the L2 cache to eight-way, then only four victim cache entries are ever used and, with a 16-way L2, the victim cache is never utilized. From this, we conclude that the victim cache can be made quite small and remain effective. In addition, our experiments have found that increasing the L2 associativity (under the optimistic assumption that changing the associativity has no impact on cycle time) has a less than 1 percent impact on performance.

7 Conclusions

For speculative parallelism with large speculative threads, unpredictable cross-thread dependences can severely limit performance. When speculation fails under an all-or-nothing approach to TLS support, the entire speculative thread must be reexecuted, thereby limiting the overall applicability of TLS to speculative threads that are either small or highly independent. To alleviate such limitations, we propose *subthreads* that allow speculative execution to tolerate unpredictable dependences between large speculative threads (that is, $> 50,000$ dynamic instructions). The support for subthreads can be implemented through simple extensions to the previously proposed TLS hardware. Using commercial database transactions, we demonstrated that subthreads can be an important part of an iterative approach to tuning the performance of speculative parallelization. In particular, we showed that subthreads can be used to reduce transaction latency, speeding up three of the five TPC-C transactions considered by a factor of 1.9 to 2.9. We also explored how to best break speculative threads into subthreads and found that having hardware support for eight subthread contexts—where each subthread executes roughly 5,000 dynamic instructions—performed best on the average. Given the large performance gains offered by subthreads for an important commercial workload, we recommend that they be incorporated into future TLS designs.

References

[1] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi, "Speculative Versioning Cache," *Proc. Fourth Int'l Symp. High-Performance Computer Architecture (HPCA '98),* Feb. 1998.

[2] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukotun, "The Stanford Hydra CMP," *IEEE Micro Magazine,* Mar.-Apr. 2000.

[3] J. Steffan, C. Colohan, A. Zhai, and T. Mowry, "A Scalable Approach to Thread-Level Speculation," *Proc. 27th Int'l Symp. Computer Architecture (ISCA '00),* June 2000.

[4] M. Tremblay, "MAJC: Microprocessor Architecture for Java Computing," *Proc. Hot Chips Conf.,* Aug. 1999.

[5] Y. Zhang, L. Rauchwerger, and J. Torrellas, "Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors," *Proc. Fifth Int'l Symp. High-Performance Computer Architecture (HPCA '99),* pp. 135-141, Jan. 1999.

[6] T. Vijaykumar, "Compiling for the Multiscalar Architecture," PhD dissertation, Univ. of Wisconsin-Madison, Jan. 1998.

[7] A. Zhai, C. Colohan, J. Steffan, and T. Mowry, "Compiler Optimization of Scalar Value Communication between Speculative Threads," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '02),* Oct. 2002.

[8] C. Colohan, A. Ailamaki, J. Steffan, and T. Mowry, "Optimistic Intra-Transaction Parallelism on Chip Multiprocessors," *Proc. 31st Int'l Conf. Very Large Data Bases (VLDB '05),* Aug. 2005.

[9] M. Prabhu and K. Olukotun, "Using Thread-Level Speculation to Simplify Manual Parallelization," *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '03),* June 2003.

[10] Standard Performance Evaluation Corp., *The SPEC Benchmark Suite,* http://www.specbench.org, 2000.

[11] J. Steffan, C. Colohan, A. Zhai, and T. Mowry, "Improving Value Communication for Thread-Level Speculation," *Proc. Eighth Int'l Symp. High-Performance Computer Architecture (HPCA '02),* Feb. 2002.

[12] M. Garzarán, M. Prvulovic, J. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas, "Tradeoffs in Buffering Memory State for Thread-Level Speculation in Multiprocessors," *Proc. Ninth Int'l Symp. High-Performance Computer Architecture (HPCA '03),* Feb. 2003.

[13] D.T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter, "Improving Preemptive Prioritization via Statistical Characterization of OLTP Locking," *Proc. IEEE Int'l Conf. Data Eng.,* 2005.

[14] J. Gray, *The Benchmark Handbook for Transaction Processing Systems.* Morgan Kaufmann, 1993.

[15] M. Franklin and G. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References," *IEEE Trans. Computers,* vol. 45, no. 5, May 1996.

[16] M. Prvulovic, M.J. Garzarán, L. Rauchwerger, and J. Torrellas, "Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization," *Proc. 28th Int'l Symp. Computer Architecture (ISCA '01),* June 2001.

[17] M. Cintra, J. Martínez, and J. Torrellas, "Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors," *Proc. 27th Int'l Symp. Computer Architecture (ISCA '00),* June 2000.

[18] H. Akkary, R. Rajwar, and S.T. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," *Proc. 36th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* Dec. 2003.

[19] A. Cristal, D. Ortega, J. Llosa, and M. Valero, "Out-of-Order Commit Processors," *Proc. 10th Int'l Symp. High-Performance Computer Architecture (HPCA '04),* Feb. 2004.

[20] J. Martínez, J. Renau, M.C. Huang, M. Prvulovic, and J. Torrellas, "Cherry: Checkpointed Early Resource Recycling in Out-of-Order Microprocessors," *Proc. 35th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* Nov. 2002.

[21] N. Tuck and D.M. Tullsen, "Multithreaded Value Prediction," *Proc. 11th Int'l Symp. High-Performance Computer Architecture (HPCA '05),* Feb. 2005.
[22] S. Sarangi, W. Liu, J. Torrellas, and Y. Zhou, "Reslice: Selective Re-Execution of Long-Retired Misspeculated Instructions Using Forward Slicing," *Proc. 38th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* Nov. 2005.
[23] M. Cintra, J. Martínez, and J. Torrellas, "Learning Cross-Thread Violations in Speculative Parallelization for Scalar Multiprocessors," *Proc. Eighth Int'l Symp. High-Performance Computer Architecture (HPCA '02),* Feb. 2002.
[24] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi, "Dynamic Speculation and Synchronization of Data Dependences," *Proc. 24th Int'l Symp. Computer Architecture (ISCA '97),* June 1997.
[25] L. Hammond, V. Wong, M. Chen, B.D. Carlstrom, J.D. Davis, B. Hertzberg, M.K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency," *Proc. 31st Int'l Symp. Computer Architecture (ISCA '04),* June 2004.
[26] L. Hammond, B.D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun, "Programming with Transactional Coherence and Consistency (TCC)," *Proc. 11th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '04),* Oct. 2004.
[27] G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar Processors," *Proc. 22nd Int'l Symp. Computer Architecture (ISCA '95),* June 1995.
[28] A. Lai and B. Falsafi, "Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction," *Proc. 27th Int'l Symp. Computer Architecture (ISCA '00),* June 2000.
[29] J.G. Steffan, C.B. Colohan, A. Zhai, and T.C. Mowry, "The Stampede Approach to Thread-Level Speculation," *ACM Trans. Computer Systems,* vol. 23, no. 3, 2005.
[30] J. Steffan and T. Mowry, "The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization," *Proc. Fourth Int'l Symp. High-Performance Computer Architecture (HPCA '98),* Feb. 1998.
[31] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proc. 17th Int'l Symp. Computer Architecture (ISCA '90),* pp. 364-373, May 1990.
[32] C. Colohan, A. Ailamaki, J. Steffan, and T. Mowry, "Extending Thread Level Speculation Hardware Support to Large Epochs: Databases and Beyond," Technical Report CMU-CS-05-109, School of Computer Science, Carnegie Mellon Univ., Mar. 2005.
[33] S. Goldstein, K. Schauser, and D. Culler, "Lazy Threads: Implementing a Fast Parallel Call," *J. Parallel and Distributed Computing,* vol. 37, no. 1, pp. 5-20, Aug. 1996.
[34] J. Steffan, C. Colohan, and T. Mowry, "Architectural Support for Threadlevel Data Speculation," Technical Report CMU-CS-97-188, School of Computer Science, Carnegie Mellon Univ., Nov. 1997.
[35] K. Keeton, "Computer Architecture Support for Database Applications," PhD dissertation, Univ. of California, Berkeley, July 1999.
[36] K. Yeager, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro,* Apr. 1996.
[37] S. Fung and J.G. Steffan, "Improving Cache Locality for Thread-Level Speculation," *Proc. IEEE Int'l Parallel and Distributed Processing Symp.,* 2006.

Christopher B. Colohan received the BASc degree in computer engineering from the University of Toronto in 1997, the MSc degree in computer science from Carnegie Mellon University in 1999, and the PhD degree from Carnegie Mellon University in 2005. The bulk of his contributions to this paper were done during his studies at Carnegie Mellon. He is working as a software engineer at Google Inc. His research is in programming and building compute clusters that process very large amounts of data.

Anastasia Ailamaki received the BSc degree in computer engineering from the Polytechnic School of the University of Patra, Greece, the MSc degrees from the Technical University of Crete, Greece, and from the University of Rochester, New York, and the PhD degree in computer science from the University of Wisconsin-Madison. In 2001, she joined the Computer Science Department at Carnegie Mellon University as an assistant professor. Her research interests are in the broad area of database systems and applications with emphasis on database system behavior on modern processor hardware and disks. Her projects at Carnegie Mellon (including staged database systems, cache-resident data bases, and the Fates Storage Manager) aim at building systems to strengthen the interaction between the database software and the underlying hardware and I/O devices. In addition, she is working on an automated schema design and computational database support for scientific applications, storage device modeling, and performance prediction, as well as Internet query caching. She is a member of the IEEE Computer Society.

J. Gregory Steffan received the BASc and MASc degrees in computer engineering from the University of Toronto in 1995 and 1997, respectively, and the PhD degree in computer science from Carnegie Mellon University in 2003. He is currently an assistant professor of computer engineering at the University of Toronto. He has also worked in the processor architecture groups of MIPS and Compaq (ALPHA). His research interests include computer architecture and compilers, distributed and parallel systems, and reconfigurable computing. The long-term goal of his research is to ease the extraction of parallelism from sequential programs so that nonexpert programmers can exploit parallel hardware such as chip-multiprocessors (CMPs) and field-programmable gate arrays (FPGAs). He is a member of the IEEE and the IEEE Computer Society.

Todd C. Mowry received the MSEE and PhD degrees from Stanford University in 1989 and 1994, respectively. He is an associate professor in the School of Computer Science at Carnegie Mellon University and he is also currently the director of the Intel Research Pittsburgh Laboratory. From 1994 through 1997, he was an assistant professor in the Electrical and Computer Engineering (ECE) and Computer Science (CS) Departments at the University of Toronto prior to joining Carnegie Mellon University in July 1997. He currently coleads the Claytronics, Log-Based Architectures, and S3 projects. He serves on the editorial board for the *ACM Transactions on Computer Systems*.