Tolerating Dependences Between Large Speculative Threads Via Sub-Threads

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

[⋆]*School of Computer Science*
Carnegie Mellon University
{natassa,tcm}@cs.cmu.edu

[†]*Department of Electrical &*
Computer Engineering
University of Toronto
steffan@eecg.toronto.edu

[∞]*Google, Inc.*
chris@colohan.com

[‡]*Intel Research Pittsburgh*

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 have minimal dependences between them. Recent work has shown that TLS can offer compelling performance improvements for database workloads, but only 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, 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 hardware support for large speculative threads that integrates several previous proposals for TLS hardware. We also introduce support for sub-threads*: a mechanism for tolerating cross-thread data dependences by checkpointing speculative execution. When speculation fails due to a violated data dependence, with sub-threads the failed thread need only rewind to the checkpoint of the appropriate sub-thread rather than rewinding to the start of execution; this significantly reduces the cost of mis-speculation. We evaluate our hardware support for large and dependent speculative threads in the database domain and find that the transaction response time for three of the five transactions from TPC-C (on a simulated 4-processor chip-multiprocessor) speedup by a factor of 1.9 to 2.9.*

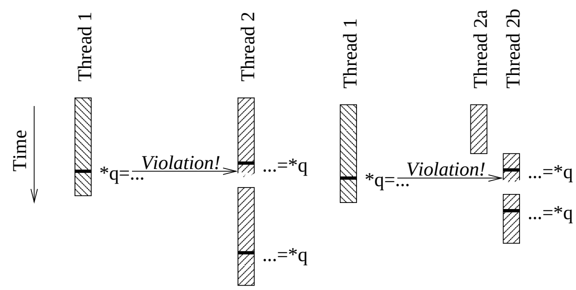
1. Introduction

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) [9, 12, 23, 25, 30], that enables either a compiler [27, 29] or the programmer [5, 18] to optimistically and incrementally introduce parallelism into a program while always maintaining functional correctness.

While 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 [9, 12, 23, 25, 30], 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 [22]) where the TLS threads are of modest size: typically a few hundred to a few thousand dynamic instructions per thread [18, 24, 27], or on benchmarks with very infrequent data dependences [8]. In addition, most of the mechanisms for supporting TLS that have been proposed to date take an *all-or-nothing* approach [9, 12, 23, 25, 30] in the sense that a given thread only succeeds in improving overall performance if it suffers *zero* inter-thread dependence violations. In the case of a single violated dependence, the speculative thread is restarted at the beginning. While 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 inter-thread dependence violations to occur far more frequently. With large threads and frequent violations the all-or-nothing approach results in very little performance gain.

In a recent paper [5] 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: (i) some transactions are latency sensitive, such as financial transactions in stock markets; and (ii) reducing the latency of transactions which 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 [16]. In the paper we showed that TLS support facilitated a nearly *twofold speedup* on a simulated 4-way chip multiprocessor for NEW ORDER, the transaction that accounts for almost half of the TPC-C workload [10], as illustrated later in Section 4.

[*] Work was performed while author was at Carnegie Mellon University.



(a) All-or-nothing TLS execution.

(b) Sub-threads divides thread 2 into thread 2a and 2b.

Figure 1. Sub-threads reduce the number of rewound instructions on a violation.

To achieve such an impressive speedup we first needed to overcome new challenges that did not arise for smaller programs such as the SPEC [22] benchmarks. In particular, after breaking up the TPC-C transactions to exploit the most promising sources of parallelism in the SQL 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 inter-thread data dependences due to internal database structures (i.e., not the SQL code itself) than in previous TLS studies. As a result, we observed no speedup on a conventional all-or-nothing TLS architecture.

1.1. Sub-Threads: Reducing Recovery Cost through Checkpointing

The primary problem with the all-or-nothing TLS architecture is that a dependence violation causes the entire speculative thread to restart, instead of just re-executing the instructions which mis-speculated. To mitigate this problem, we propose the use of *sub-threads*. A sub-thread is started by creating a lightweight checkpoint of the speculative thread. When a dependence violation is detected, the speculative thread rewinds to the start of the sub-thread which contained the dependent load, instead of rewinding the entire speculative thread. Figure 1 shows how sub-threads can improve performance by avoiding a complete rewind.

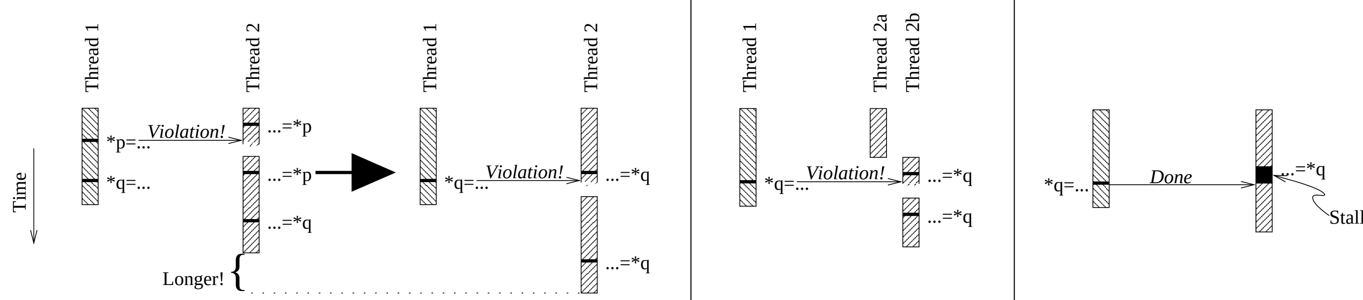
Support for sub-threads make TLS more useful, since it enables the programmer to treat the parallelization of a program as a *performance tuning* process. Without sub-threads, TLS only provides a performance gain if a chosen thread decomposition results in speculative threads with infrequent data dependences between them. With sub-threads, the programmer can engage in an iterative process: (i) decompose the program into speculative threads; (ii) profile execution and observe which data dependence causes the most execution to be rewound; (iii) modify the program to avoid this data dependence; and (iv) iterate on this process (by returning to step (ii)).

Without sub-thread support, removing a data dependence can actually *degrade performance*: as shown in Figure 2(a), removing the early dependence (through `*p`) only delays the inevitable re-execution of the entire speculative thread because of the later dependence (through `*q`). With sub-thread support—and an appropriate choice of sub-thread boundaries—each removed dependence will gradually improve performance (Figure 2(b)). With enough sub-threads per thread, TLS execution approximates an idealized parallel execution (Figure 2(c)) where parallelism is limited only by data dependences, effectively stalling each dependent load until the correct value is produced. In summary, sub-threads allow TLS to improve performance even when speculative threads have unpredictable and frequent data dependences between them.

1.2. Related Work

Sub-threads 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 [1, 6], and a single checkpoint in the cache [15]. Martínez's checkpointing scheme [15] effectively enlarges the reorder buffer and is also integrated with TLS, and is thus the most closely related work. The sub-thread design we present in this paper could be used to provide a superset of the features in Martínez's work at a higher cost: sub-threads 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 [26]—the techniques we use to create sub-threads could also be used to create checkpoints at high-confidence prediction points following Tuck and Tullsen's method.

For TLS, tolerating failed speculation using sub-threads implies tolerating data dependences between speculative threads. A recent alternative approach for tolerating cross-thread dependences is *selective re-execution* [20], where only the slice of instructions involved in a violated dependence is re-executed. Other studies have explored predicting and learning data dependences and turning them into synchronization [3, 17, 24], or have used the compiler to mark likely dependent loads and tried to predict the consumed values at run time [24]. Initially we tried to use an aggressive dependence predictor like proposed by Moshovos [17], but found that only one of 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 call-



(a) Eliminating the dependence through `*p` can potentially *hurt* performance with all-or-nothing TLS execution.

(b) Execution with sub-threads.

(c) Idealized parallel execution.

Figure 2. The use of sub-threads enables a performance tuning process, where eliminating data dependences improves performance.

ing context. Support for sub-threads provides a more elegant solution which is *complementary* to hardware based prediction and software based synchronization techniques, since using sub-threads significantly reduces the high cost of mis-speculation.

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 [12, 18]. 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 thread, as well as allowing partial rollback of a thread through sub-threads. 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 [13], but requiring changes to the programming model to accommodate that new architecture [11]. The architecture presented in this paper is closer to existing chip-multiprocessor architectures, and can execute both regular binaries as well as those which have been modified to take advantage of TLS.

1.3. Contributions

The primary contribution of this paper is that it introduces sub-threads, a mechanism to tolerate data dependences between large speculative threads. This support allows us to evaluate the speculative parallelization of individual commercial database transactions. However, to parallelize these large dependent speculative threads requires more than just sub-thread support: we must buffer a large amount of speculative state, tolerate dependences between threads through aggressive update propagation, provide hardware mechanisms to profile data dependences, and provide mechanisms to help programmers remove critical dependences from software. These techniques have been partially or fully demonstrated in isolation in previous work; in this paper we present a unified design which incorporates all of these features, together with support for sub-threads. We believe that the proposed hardware can be used to support large and dependent speculative threads in other application domains as well, expanding the scope for TLS.

2. Supporting Large Speculative Threads

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 which is *violated* must be *detected*, and *corrected* by re-starting the offending thread. Hardware support for TLS makes the detection of violations and the restarting of threads inexpensive [12, 19, 21, 23].

Our database benchmarks stress TLS hardware support in new ways which have not been previously studied, since the threads are necessarily so much larger. Previous work has studied threads with various size ranges, including 3.9–957.8 dynamic instructions [27], 140–7735 dynamic instructions [18], 30.8–2,252.7 dynamic instructions [24], and up to 3,900–103,300 dynamic instructions [8]. 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 3KB to 35KB; additional state is required to store multiple versions of cache lines for sub-threads. 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 which cannot be easily synchronized and forwarded by the compiler [29], 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

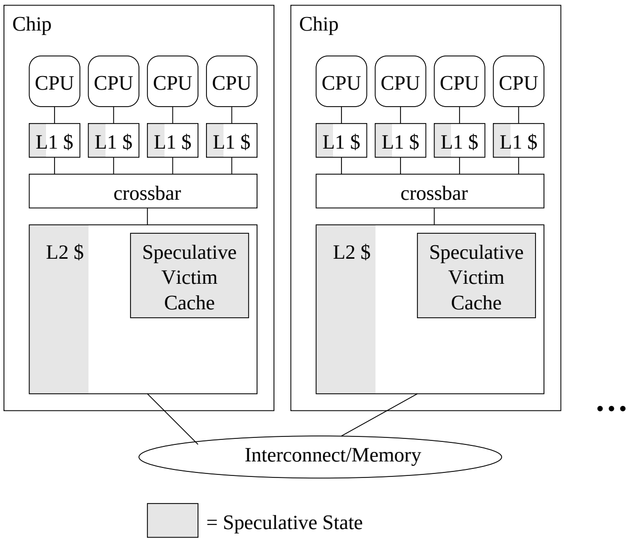


Figure 3. An overview of the CMP architecture that we target, and how it is extended to support TLS.

tolerating data dependences between threads attractive.

2.1. Buffering Speculative State

Previous work on TLS assumes that the speculative state of a thread fits in either speculative buffers [12, 21] or in the L1 cache [2, 9, 23]. Given the large amount of speculative state per thread generated by our database workloads, 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 [19]; and Cintra *et. al.* proposed a hierarchical TLS implementation which 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) [2]. In this paper, we propose a similar hierarchical implementation, with one important difference from Cintra's scheme: it allows *all* threads to store their speculative state in the larger L2 caches. With this support (i) all threads can take advantage of the large size of the L2 cache, (ii) threads can aggressively propagate updates to other more recent threads, and (iii) we can more easily implement *sub-threads*, described in later in Section 2.2.

As shown in Figure 3, we assume an underlying CMP where each core has a private L1 cache, and multiple cores share a single chip-wide L2 cache. For simplicity, in this paper we assume that each CPU is capable of executing only a single thread at a time (i.e., no SMT execution), and we consider speculative execution within a single CMP only (although our scheme could be extended to beyond a chip [23]). Both the L1 and L2 caches maintain speculative state: each L1 cache buffers cache lines that have been speculatively read or modified by the thread executing on the corresponding CPU, while the L2 caches maintain inclusion and buffer copies of all speculative cache lines that are in any L1 cache. The L2 cache tracks which cache lines have been speculatively loaded by each thread within the chip, and which words have been speculatively modified within each cache line—i.e., speculative loaded state is tracked at a cache line granularity while speculative modified state is tracked at a word granularity. This speculative state is used to detect dependence violations, and to commit or discard speculative state appropriately. The L1 caches are write-through, ensuring that store values are aggressively propagated to the L2 where dependent threads may load those values to avoid dependence violations. Since multiple threads may modify the same cache line in the L2 cache, we allow the L2 cache to manage multiple versions of each cache line by using the different "ways" of each associative set. Furthermore, we add a 64-entry victim cache to the L2 to catch any speculative cache lines which are evicted from the regular L2 cache.[1]

In the L2 cache we require 2 bits of storage per cache line per sub-thread tracked. This is equivalent to requiring a speculative thread context per sub-thread. For a system with 4 CPUs (or hyperthreads) sharing a cache, running 1 speculative thread per CPU (hyperthread), and 4 sub-threads per speculative thread, this would mean an additional 16 bits of storage per cache line.

In summary, this design supports 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 [4].

2.2. Tolerating Dependences with Sub-Threads

The motivation for sub-threads 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* [17, 24]. A data dependence predictor automatically synchronizes a dependent store/load pair to avoid a dependence violation, while a value predictor provides a predicted value to the load which can then proceed independently from the corresponding

[1] We choose a 64-entry victim cache because it is large enough to avoid stalling threads due to cache overflows for our worst case: our largest transaction with a 4-way set associative 2MB L2 cache with 8 sub-threads per thread. A smaller victim cache would likely be sufficient for the common case.

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 sub-threads are *complimentary* to prediction, since the use of sub-threads reduces the penalty of any mis-prediction.

Sub-threads 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 sub-threads for each of four speculative threads (eight sub-threads total), then the L2 cache must be extended to maintain speculative state for eight distinct thread contexts. At the start of a new sub-thread, the register file is backed up[2], and all subsequent speculative state is saved in the next thread context. New sub-threads 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 sub-threads are supported, any violation will specify both the thread *and sub-thread* that needs to be restarted. Within a speculative thread, the sub-threads 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 sub-threads: dependence tracking between sub-threads 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 sub-threads, 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 sub-thread 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 incorrect values generated by the primary violated thread, all of these later threads must also be restarted through a *secondary violation*. With support for sub-threads we can do better, as shown in Figure 4. In Figure 4(a) when the dependence violation between thread *1* and sub-thread *2b* is detected, all of thread *3* and *4*'s sub-threads are restarted through secondary violations even though sub-threads *3a* and *4a* completed before sub-thread *2b* started. Since sub-threads *3a* and *4a* could *not* have consumed any data from *2b*, it is unnecessary to restart them. Hence we prefer the execution in shown in Figure 4(b), which requires that we track the temporal relationship between sub-threads by having every speculative thread maintain a *sub-thread start table* as follows. When a sub-thread begins, it sends a *subThreadStart* message to all logically-later threads. On receipt of a *subThreadStart* message, each thread records the identifier of its currently-executing sub-thread in the table-entry for the sub-thread that sent the message. As illustrated in Figure 4(b), when the secondary violation is originated by thread *2b*, the later threads *3* and *4* consult their sub-thread start table entries for thread *2b* to find that they need to restart sub-threads *3b* and *4b* respectively. Hence this support provides the more selective restarting behavior that we desire.

Later in Section 5.1 we investigate the trade-offs in the frequency of starting new sub-threads, and the total number of sub-threads supported. Next we look at how our support for large speculative threads and sub-threads leads to an elegant model for iterative feedback-driven parallelization.

[2]While we do not model a particular implementation of register file back-up, this could be accomplished quickly through shadow register files, or more slowly by backing up to memory.

3. Exploiting Sub-Threads: 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 inter-thread 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 extra speed is obtained by modifying only the critical regions of code. To support this iterative parallelization process we require hardware to provide profile feedback reporting which store/load pairs triggered the most harmful violations—those that caused the largest amount of failed speculation.

3.1. Profiling Violated Inter-Thread Dependences

Hardware support for profiling inter-thread dependences requires only a few extensions to basic TLS hardware support. Each processor must maintain an *exposed load table* [24]—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-thread 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-thread. 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

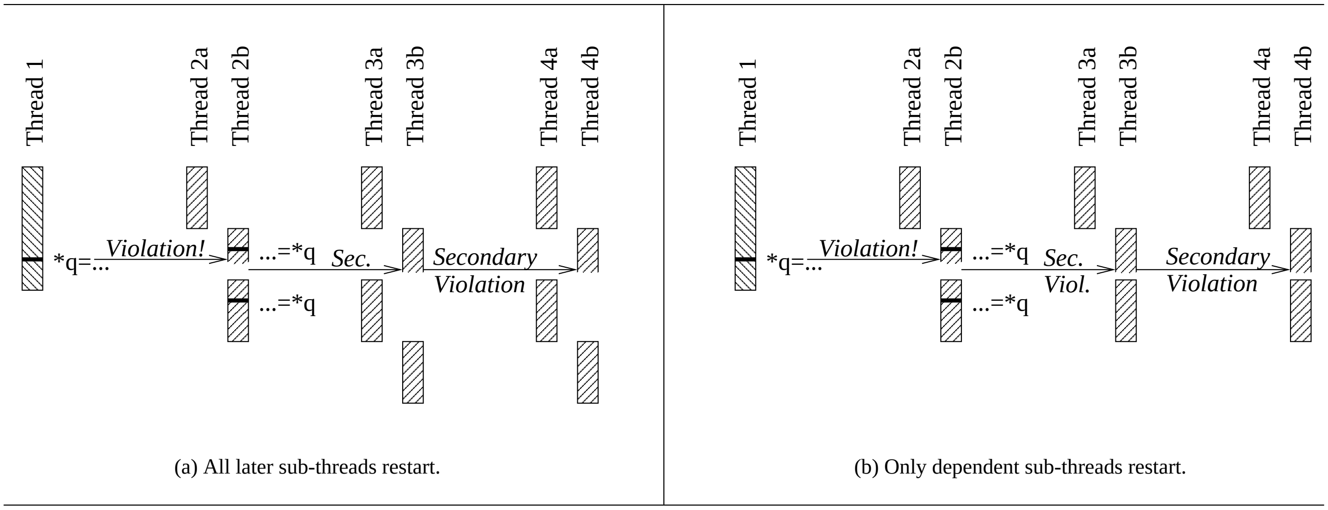


Figure 4. The effect of secondary violations with and without sub-thread dependence tracking.

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-thread 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.

3.2. Tuning Parallelized Transactions

Previous work [5] 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. 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), but more importantly removes dependences from the critical path.

3.3. When to Use TLS

TLS uses multiple CPUs to improve the performance of a single transaction. To optimize complete system performance, the DBMS must decide when to use TLS. If CPUs are otherwise idle (such as when transaction is holding a heavily contended lock or the system is bottlenecked on the network or disk) then the idle CPUs can be used for TLS. TLS also can be used to improve the performance of high priority transactions at the expense of lower priority transactions. When more transactions are available to be run than CPUs are available then TLS should be applied less aggressively.

4. Experimental Setup

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

4.1. Benchmark Infrastructure

Our experimental workload is composed of the five transactions from TPC-C (NEW ORDER, DELIVERY, STOCK LEVEL, PAYMENT and ORDER STATUS).[3] 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

[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-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.

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

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 (100MB) buffer pool.[4] 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: (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 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 [5] we described how we iteratively optimized the database system for TLS using the methodology described in Section 3. We evaluate the hardware using fully optimized benchmarks.

4.2. Simulation Infrastructure

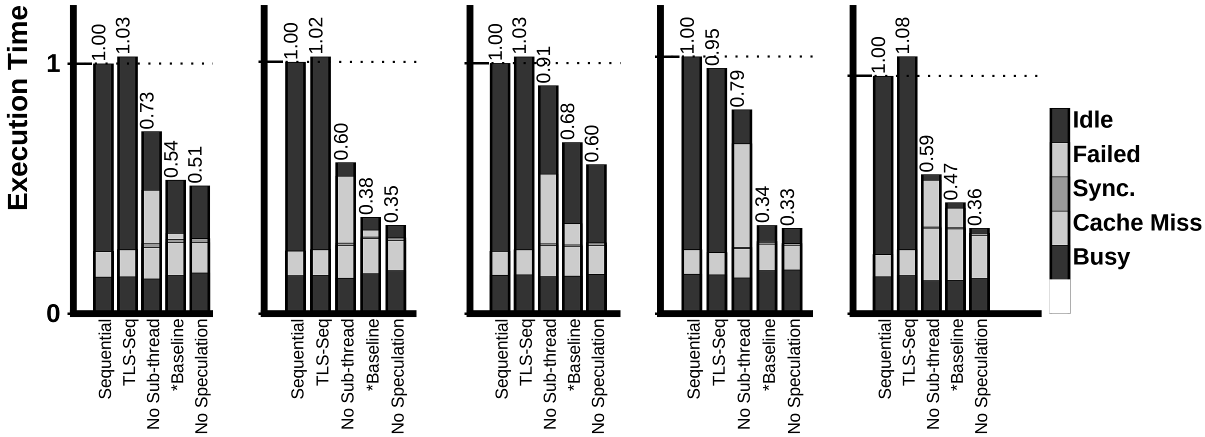
We perform our evaluation using a detailed, trace-driven simulation of a chip-multiprocessor composed of 4-way issue, out-of-order, superscalar processors similar to the MIPS R14000 [28], but modernized to have a 128-entry reorder buffer. Each processor has its own physically private data and instruction 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 and contention) are all modeled, and are parameterized as shown in Table 1. For now, we model a zero-cycle latency for the creation of a sub-thread, including register back-up. This is a user-level simulation: all instructions in the transactions and DBMS are simulated, while 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 main memory.

4.3. Benchmark Characterization

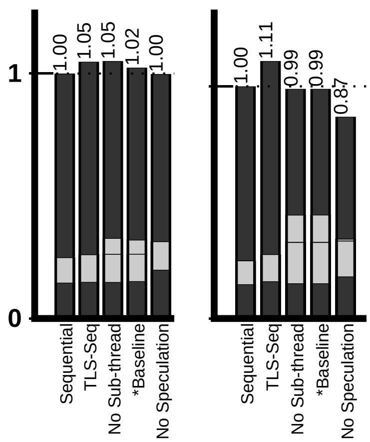
We start by characterizing the benchmarks themselves, so we can better understand them. As a starting point for comparison, we run each original sequential benchmark, which shows the execution time with no TLS instructions or any other software transformations running on one CPU of the machine (which has 4 CPUs). This SEQUENTIAL experiment takes between 17 and 509 million cycles (Table 2) to execute, but this time is normalized to 1.0 in Figure 5—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 threads, and also due to the changes to the DBMS we made to parallelize it. The TLS-SEQ experiment in Figure 5 shows the performance of this parallelized executable running on a single CPU—the additional software overhead is reasonable, impacting performance by a factor of 0.93 to 1.05. The speedup of 1.05 is the result of our added code inadvertently enabling further compiler optimization.

When we apply TLS without sub-thread support on a 4-CPU system, shown in the NO SUB-THREAD bars, performance improves for the majority of 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 4 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% coverage—in DELIVERY OUTER the outer loop of the transaction is parallelized, which has 99% coverage, but increases the average thread size from 33,000 dynamic instructions to 490,000 dynamic instructions. With larger threads the penalty for mis-speculation is much higher, and this shows up as a much larger *Failed* component in the NO SUB-THREAD bar of the DELIVERY OUTER graph. The PAYMENT and ORDER STATUS transactions do not im-

[4]This is roughly the size of the entire dataset for a single warehouse. This ensures that reads will not go to disk, data updates are performed at transaction commit time, consume less than 24% of execution time [14], and can be executed in parallel with other transactions.



(a) NEW ORDER (b) NEW ORDER 150 (c) DELIVERY (d) DELIVERY OUTER (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-thread	Baseline execution with sub-thread support disabled.
Baseline	Execution on hardware described in this paper.
No Speculation	Upper bound—modified hardware to treat all speculative writes as non-speculative.

Category	Explanation
Idle	Not enough threads were available to keep the CPUs busy.
Failed	CPU executed code which was later undone due to a violation (includes all time spent executing failed code.)
Synchronization	Stalled awaiting synchronization during escaped speculation.
Cache Miss	Stalled on a cache miss.
Busy	CPU was busy executing code.

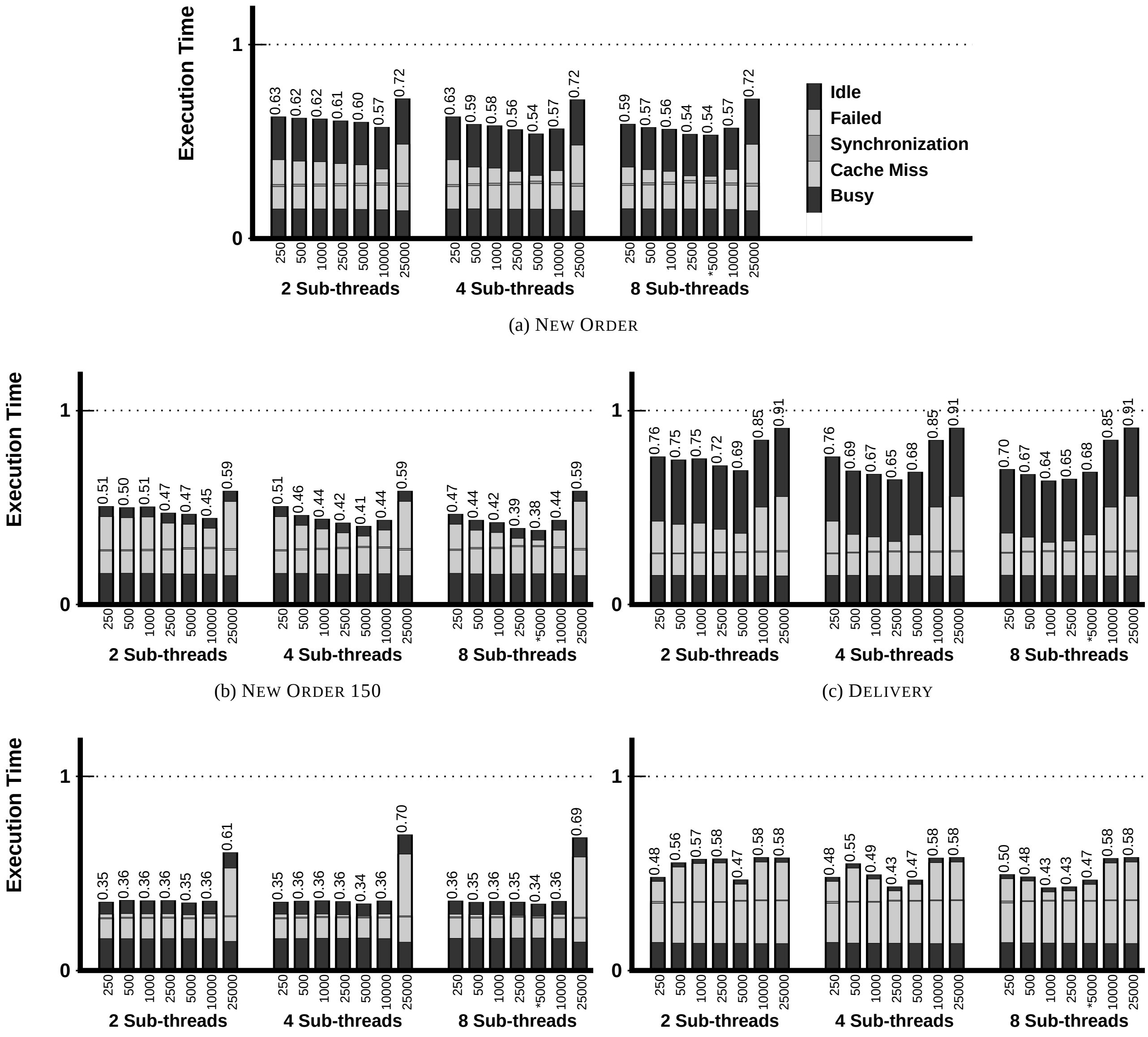
Figure 5. Overall performance of optimized benchmarks on a 4-CPU system.

prove with TLS since they lack significant parallelism in the transaction code, and so we omit them from further discussion. By parallelizing the execution over 4-CPUs we also execute using 4 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 non-speculative and hence ignoring all data dependences between threads. This execution shows sub-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 the spreading of data across four caches [7]. 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.

5. Experimental Results

The BASELINE experiment in Figure 5 shows TLS execution with 8 sub-threads per speculative thread, and 5000 instructions per sub-thread. Sub-thread support is clearly beneficial, achieving a speedup of 1.9 to 2.9 for three of the five transactions. Sub-threads 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



(a) NEW ORDER

(b) NEW ORDER 150

(c) DELIVERY

(d) DELIVERY OUTER

(e) STOCK LEVEL

Figure 6. Performance of optimized benchmarks on a 4-CPU system when varying the number of supported sub-threads per thread from 2 to 8, varying the number of speculative instructions per sub-thread from 250 to 25000. The BASELINE experiment has 8 sub-threads and 5000 speculative instructions per thread.

ORDER STATUS it appears in the graph that there is still room for improvement, but hand-analysis determined that the remaining time spent on failed speculation is due to actual data dependences which are difficult to optimize away in the code. The improvement due to sub-threads is most dramatic when the speculative threads are largest: the DELIVERY OUTER benchmark executes more than twice as quickly with sub-thread support enabled than without. Sub-threads do not have a large impact on the time spent servicing cache misses: this shows that the additional cache state required to support sub-threads does not exceed the capacity of the L2 cache.

5.1. Choosing Sub-thread Boundaries

Sub-threads allow us to limit the amount of execution rewound on a mis-speculation; but how frequently should we start a new sub-thread, and how many sub threads are necessary for good performance? We want to minimize the number of sub-thread contexts supported in hardware, and we also want to understand the performance benefits of increasing the number of contexts. Since inter-thread dependences

Table 2. Benchmark statistics.

Benchmark	Sequential Exec. Time (Mcycles)	Coverage	Average Thread Stats: Size (Dyn. Instrs.)	Spec. Insts. per Thread	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

(and hence violations) are rooted at loads from memory, we want to start new sub-threads 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 sub-threads 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 [24]. Second, if the load does cause a violation, how much correct execution will the sub-thread avoid rewinding? If the load is near the start of the thread or a previous sub-thread, then a violation incurred at this point will have a minimal impact on performance. Instead, we would rather save the valuable sub-thread context for a more troublesome load. A simple strategy that works well in practice is to start a new sub-thread every nth speculative instruction—however, the key is to choose n carefully.

In Figure 6 we show an experiment where we varied the number of sub-threads available to the hardware, and varied the spacing between sub-thread start points. We would expect that the best performance would be obtained if the use of sub-threads is conservative, since this minimizes the number of replicate versions of each speculative cache line, and hence minimizes cache pressure. Since each sub-thread requires a hardware thread context, using a small number of sub-threads also reduces the amount of required hardware. A sub-thread would ideally start just before the first mis-speculating instruction in a thread, so that when a violation occurs the machine rewinds no farther than required.

If the hardware could predict the first dependence very accurately, then supporting 2 sub-threads per thread would be sufficient. With 2 sub-threads the first sub-thread 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 sub-threads periodically as the thread executes.

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

When we initially chose a distance between sub-threads 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-threads per thread. This value has proven to work remarkably well for all of our transactions. Closer inspection of both the thread sizes listed in Table 2 and the graphs of Figure 6 reveals that instead of choosing a single fixed sub-thread size, a better strategy may be to customize the sub-thread size such that the average thread size for an application would be divided evenly into sub-threads.

One interesting case is DELIVERY OUTER, in Figure 6(d), where a data dependence early in the thread's execution causes all but the non-speculative thread to restart. With small sub-threads the restart modifies the timing of the thread's execution such that a data dependence much later in the thread's execution occurs in-order, avoiding violations. Without sub-threads, or with very large sub-threads (such as the 25000 case in Figure 6(d)) this secondary benefit of sub-threads does not occur.

6. 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 re-executed; thereby limiting the overall applicability of TLS to speculative threads that are either small or highly independent. To alleviate such limitations, we propose *sub-threads* that allow speculative execution to tolerate unpredictable dependences between large speculative threads (i.e., $>$50,000 dynamic instructions). Support for sub-threads can be implemented through simple extensions to previously proposed TLS hardware. Using commercial database transactions, we demonstrated that sub-threads can be an important part of an iterative approach to tuning the performance of speculative parallelization. In particular, we showed that sub-threads 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 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. Given the large performance gains offered by sub-threads for an important commercial workload, we recommend that they be incorporated into future TLS designs.

References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *MICRO 36*, December 2003.

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

[3] M. Cintra and J. Torrellas. Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors. In *HPCA 8*, February 2002.

[4] 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 University, March 2005.

[5] C. Colohan, A. Ailamaki, J. Steffan, and T. Mowry. Optimistic Intra-Transaction Parallelism on Chip Multiprocessors. In *VLDB 31*, August 2005.

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

[7] S. Fung and J. G. Steffan. Improving cache locality for thread-level speculation. In *IPDPS 20*, April 2006.

[8] M. Garzarán, M. Prvulovic, J. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. In *HPCA 9*, February 2003.

[9] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative versioning cache. In *HPCA 4*, February 1998.

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

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

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

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

[14] K. Keeton. *Computer Architecture Support for Database Applications*. PhD thesis, University of California at Berkeley, July 1999.

[15] J. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *MICRO 35*, November 2002.

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

[17] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic speculation and synchronization of data dependences. In *ISCA 24*, June 1997.

[18] M. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *PPoPP '03*, June 2003.

[19] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas. Removing architectural bottlenecks to the scalability of speculative parallelization. In *ISCA 28*, June 2001.

[20] S. Sarangi, W. Liu, J. Torrellas, and Y. Zhou. Reslice: Selective re-execution of long-retired misspeculated instructions using forward slicing. In *MICRO 51*, November 2005.

[21] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *ISCA 22*, June 1995.

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

[23] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. A scalable approach to thread-level speculation. In *ISCA 27*, June 2000.

[24] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. Improving value communication for thread-level speculation. In *HPCA 8*, February 2002.

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

[26] N. Tuck and D. M. Tullsen. Multithreaded value prediction. In *HPCA 11*, February 2005.

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

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

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

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