In Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT 2009).

# Mapping Out a Path from Hardware Transactional Memory to Speculative Multithreading

Leo Porter, Bumyong Choi, and Dean M. Tullsen
*University of California, San Diego*

***Abstract*— This research demonstrates that coming support for hardware transactional memory can be leveraged to significantly reduce the cost of implementing true speculative multithreading. In particular, it explores the path from eager conflict detection HTM to full support of efficient speculative multithreading, focusing on the case where frequent memory dependencies exist between speculative threads. The result is a unified memory architecture capable of effective support for transactional parallel workloads and efficient speculative multithreading.**

***Keywords*-Chip Multiprocessors, Speculative Multithreading, Transactional Memory**

## I. Introduction

Transactional Memory (TM) has been proposed as a powerful programming primitive for shared memory multiprocessors to replace traditional lock-based synchronization [1]–[12]. Prior research has examined both software transactional memory [10]–[12] and hardware transactional memory [1]–[9]. TM has gained significant interest in both academia and industry because of the potential to make programming the coming multi-core and many-core processors easier [13] and more effective [1]. These factors led to the inclusion of hardware support for transactional memory in SUN's proposed Rock Processor [14]. We expect other implementations to follow.

Speculative Multithreading (SpMT) [15]–[24] is another architectural alternative that becomes increasingly attractive as we enter the multi-core era more fully. It provides the hope of exploiting available parallel cores to increase the performance of a single thread of execution. Serial code will increasingly be a limiting factor both in terms of power and performance [25].

Given the imminent arrival of hardware support for TM, we would like to leverage that hardware to add support for true speculative multithreading, at a lower incremental cost than supporting speculative multithreading from scratch. The focus of this work is to map out a path from HTM to SpMT. What are the elements that we would need to add to a memory design that supported HTM in order to allow SpMT? How should they work in an HTM context? Which of those elements are crucial and which are not?

It has been stated and demonstrated that transactional memory provides support for speculative multithreading, or thread level speculation [4]–[6], [26], [27] via its support for opportunistic concurrency. That is, if two iterations of a loop both access shared data within a transaction, but the accesses conflict in only a small number of cases, the iterations will execute in parallel when there is no conflict, and cause transactions to abort and execute in series when the conflict exists. Thus, the traditional domain of TM is code that is truly parallel (dynamically). It does not support the parallel execution of code where frequent and numerous dependences exist. It also requires significant compiler or programmer support to identify and properly exploit opportunities for thread-level speculation.

To perform this study, we created an architecture for speculative multithreading that executes unmodified single-threaded binaries, relies on prediction for register dependence handling, and relies on the memory subsystem to handle memory dependences. We use this SpMT framework in concert with a number of HTM designs. In addition to the designs previously proposed, we examine a number of intermediate steps along the path from HTM to full SpMT support and show the performance potential at each step. We ultimately define a unique architecture which efficiently supports both TM and dependency-rich SpMT.

Assuming this speculative multithreading approach, we evaluate HTMs which feature eager conflict detection (also assuming enough architectural support to be able to execute in this environment) using SPEC CPU2000 binaries and found that a limited speedup of 10% can be obtained on two cores. The primary weaknesses of traditional HTMs in a SpMT environment are the inability to differentiate more speculative threads from less speculative threads, false sharing, cold cache effects whenever a new thread is initiated on a core, the delayed notification of potential dependencies, and the inability to forward available values when loaded by a more speculative thread. By introducing improvements to HTM with eager conflict detection, we are able to achieve more than a 3x increase in the effectiveness of SpMT (resulting in a speedup of 36% on two cores).

The improvements considered along the path from TM to SpMT include word-granularity tracking of memory coherence, support for ordered transactions, forwarding of values to higher ordered transactions, and a write-update cache coherence protocol.

This paper is organized as follows. Section II discusses related work. The assumed SpMT architecture is outlined in

Section III. The methodology is provided in Section IV. In Section V, the baseline transactional memory is discussed. Improvements to hardware TM are detailed in Section VI with associated performance results. Section VII addresses the generality of these results.

## II. Related Work

Speculative Multithreading [15]–[24] provides the ability to parallelize otherwise serial code. These proposals vary in thread identification and spawn, handling of register and memory dependences, etc.

Previous studies in the speculative multithreading literature produce speculative threads either using a compiler or hand tuning [15]–[19], statically using binary instrumentation [20], or else the speculative threads are identified entirely in hardware [22]–[24]. When most SpMT proposals encounter an inter-thread dependence conflict, the speculative thread is squashed; however, Dynamic Multithreading only squashes, then re-executes, those instructions affected by the dependence [23], [24].

Memory dependences pose a significant challenge for SpMT as they generally cannot be identified until runtime and thus require additional support. A number of SpMT memories have been proposed, starting with the Address Resolution Buffer (ARB) [15], [28]. The ARB benefits from simple memory address disambiguation, but a single centralized data cache for multiple cores may represent a bottleneck. To address this issue, multiple memories which buffer speculative state in local data caches were proposed to provide improved performance [16], [29], [30]. Our work shares many of the same goals as prior SpMT memory proposals. However, none of these models support TM code, and, of course, do not leverage a potential transactional hardware base.

Hardware Transactional Memory (HTM) was proposed by Herlihy and Moss as a simpler means of synchronization capable of performing optimistic concurrency [1]. This hardware model has been adopted by more recent TM proposals including LTM [2], UTM [2], PTM [31], LogTM [32], and VTM [8]. While these proposals vary significantly in their interaction with software transactional memory, ability to handle context-switches and transactional overflow, support for nested transactions, etc., their conflict detection semantics are similar and can all be represented for this work by a single hardware model (the ULI model, described in Section V-A). LogTM-SE [9] proposes eager conflict detection based on hashed signatures at block granularity. This reduces the storage requirements on caches to track read-set and write-set information. However, a conflict in LogTM-SE is the same as in our ULI model. TokenTM [3] avoids modifying coherence protocols by using tokens to track conflicts. Again, the notion of a conflict in TokenTM is the same as our ULI model. Although the details of SUN's proposed Rock Processor [14] are not released, the hardware support is likely to be similar to Hybrid Transactional Memory [33], again with conflict detection semantics similar to our ULI model.

FlexTM [34] separates conflict detection and conflict management. In some benchmarks, our LAZY design achieves stronger performance than our eager design. For these cases, switching between eager and lazy conflict management could be useful. However, the software overheads for FlexTM may hinder performance.

Dependence Aware TM (DATM) [35] offers some of the features recommended in this work. However, there are key differences. DATM adheres to transactional semantics which offer weaker guarantees than those required by our SpMT architecture. Most notably, stale data read by a transaction can persist in a cache and be read by a non-transaction. In addition, DATM requires bus requests for most transactional requests even in the case of a cache hit whereas our stronger designs do so only in the event of conflicts on that line. Lastly, our later designs allow previous transactions to overwrite words written by later transactions whereas DATM does not. The impact of this last limitation is discussed in Section VI. Most importantly, however, DATM and our conclusions are complementary. The features added by DATM are shown to aid transactional parallelism and the features added by our designs are shown to aid speculative parallelism. The overlap of many of these features emphasizes their importance.

Transactional Coherence and Consistency (TCC) has been proposed not only as a Transactional Memory model but as an alternative to traditional per-memory operation coherence [5], [6]. Similar to TCC, Bulk [4] performs lazy conflict detection at word granularity with write update but does so using hashed signatures rather than cache line bits. These proposals are represented by our LAZY model (Section VI-H1).

The ability of Transactional Memory to support Speculative Multithreading is mentioned in TCC [5], [6] and Bulk [4]. Transaction sizes in the context of speculative parallelism are addressed in [26]. Recent work extends programming language loop constructs to provide support for speculative multithreading [27]. Bulk also uses HTM for compiler-based SpMT. However, none of these mechanisms provide full support for the parallel execution of dependent threads.

## III. SpMT Execution Model

This section describes our SpMT execution model and the minimum architectural support needed by any of the memory designs to be able to execute in our SpMT architecture. The focus of the rest of this paper is the design of the memory architecture itself; however, this section primarily details other aspects of the architecture necessary to fully support speculative multithreading, and what assumptions were made about that architectural support.

### A. Speculative Thread Selection

Prior proposals have assumed varying degrees of compiler support [15]–[24]. We fully believe that compiler support will maximize the potential performance of speculative multithreading. However, for this study we assume no compiler or software support to identify speculative threads. Assuming no support provides a general SpMT model that allows us to examine our different memory designs in a fashion unbiased by the compiler, as we are not able to structure the code (even unintentionally) to favor one model. Additionally, this approach significantly expands the domain of SpMT – enabling parallel execution of legacy uniprocessor code, allowing the use of a single binary for a family of architectures that support different levels of SpMT, etc.

Our strict insistence on no compiler support makes our results potentially pessimistic, compared to a system that allowed compiler involvement. Even simple compiler support on single-thread binaries that moves or removes problem memory operations could greatly increase available SpMT parallelism. The bottom line, though, is that achieving the highest possible SpMT speedups is not the goal of this research. Rather, we are trying to understand the support in the transactional memory subsystem necessary to achieve the full potential of SpMT. The absolute magnitude of the speedups achieved is less significant.

Following the terminology of [36], a speculative thread is characterized by two PC addresses: the point in execution where a new thread will be created, called the Spawning Point (SP), and the point in the program where the new thread will begin executing, called the control quasi-independent point (CQIP). The new thread ends when it reaches another thread's CQIP and validates that dependences between the two threads were handled correctly. Compared to transactional execution, the entire thread essentially becomes a single transaction, and either completes execution atomically or is squashed/aborted in its entirety.

Speculative threads have been initiated at loop iterations, loop continuations, and function calls in the past [36]. The Mitosis architecture [17] can construct a SP-CQIP pair from any two arbitrary points in the program. However, without compiler support, our options are more limited. We target loop iterations and function call continuations, as they can be easily identified at run time and have been shown to be good candidates [36]. Out-of-order spawns are possible when a less speculative thread encounters a spawnpoint.

When an executing thread fetches an instruction whose address is a spawn point, we determine if an idle core is available. If one is available, we transmit any program state needed by the new thread (e.g., register live-ins) and start the speculative thread on that core after a communication and fetch delay.

When the parent thread encounters the CQIP of its child it stops fetching and waits for the CQIP to be committed. When the CQIP is committed, the child thread is validated. Validation includes the checking of any register live-ins and ensuring that the speculative thread was not squashed due to a memory dependence conflict. Speculative threads validated by non-speculative threads become non-speculative. Speculative threads wait to commit until they become non-speculative

A confidence counter is maintained to indicate the frequency at which a spawn point results in a committed or squashed speculative thread. Spawn points which frequently squash are ignored after reaching a number of consecutive failures but are still given a small chance of re-execution. We found that allowing nine consecutive failures and giving a five percent chance for retry of failed spawnpoints was successful at reducing interference from frequently squashing threads.

To maintain a desirable speculative thread length, threads greater than ten instructions and less than ten thousand instructions are targeted. This can be accomplished by recording the length of previously executed threads.

Threads may be squashed due to dependence issues (discussed below), a system call by the non-speculative thread, or when the non-speculative thread has transactional state overflow. A squash of any thread causes all more speculative threads to be squashed. Speculative threads may be paused (caused to stop execution until made non-speculative) if they encounter a system call or cause a transaction overflow.

A store miss request can be issued at execute, preemptively loading the cache line, or at commit. By requesting the line during execute, the request latency is partially hidden rather than delaying the store entirely to commit. All memory designs assume this realistic optimization. This introduces the possibility of squashes due to wrong-path write miss requests for all memory designs except those which handle conflict detection at thread commit [4]–[6].

Our simulator assumes a centralized structure for speculative thread coordination which we refer to as the Global Speculative Thread Supervisor (GSTS). Important thread information (thread ordering, CQIP of children, etc.) is distributed to each core to avoid unnecessary communication with the GSTS. In a general TM system, parallel transactions can typically commit in any order that does not produce conflicts. Because we must maintain sequential semantics, we must commit threads in execution order. Support for this already exists in some TM proposals [4]–[6]. This is different than having the memory coherence know and account for that order, which is addressed in a later section. This is a useful distinction, both because the former is necessary and the latter is an optimization, and also because the former is easily supported in the GSTS, and the latter requires coherence modification.

| | | | |
|---|---|---|---|
| Cores | 2 | I cache miss penalty | 20 cyc |
| Fetch width/core | 4 | D cache | 32k, 4 way |
| INT instruction queue | 64 entries | D cache miss penalty | 20 cyc |
| FP instruction queue | 64 entries | shared L2 cache | 2 MB, 8 way |
| Reorder Buffer entries | 128 | L2 miss penalty | 75 cyc |
| FP registers per core | 132 | L3 | 4 MB, 8 way |
| Fork penalty | 10 cyc | L3 miss penalty | 315 cyc |
| Cache line size | 64 bytes | Victim cache entries | 8 |
| I cache | 32k, 4 way | | |

**Table I: Architectural Specification**

### B. Handling Register Dependences

Because we execute an unmodified single-threaded binary, the code, although executed in parallel, assumes a single unified register file. As a result, there will be register dependences that cross thread boundaries. Prior work has managed these dependences in various ways, from explicitly forwarding [15], [37], [38], to hardware prediction [36], to software precomputation [17]. In this research we initially want to decouple the register dependence problem (which is orthogonal to the memory design) and thus adopt a very simple model for our initial results – register dependences are handled by prediction and all predictions are correct. We consider the case of a realistic, but not particularly aggressive, predictor in Section VII and show that all the key conclusions of the paper still hold. . We should note that the assumption of perfect or near-perfect prediction is not an absurd one – the Mitosis register prediction model [17], because it leverages code from the spawning thread, can be made to be arbitrarily accurate. However, this causes a larger delay in thread spawn.

## IV. Methodology

To evaluate the performance of our various models of speculative multithreading, we added support for SpMT, including the full suite of memory designs, to the SMTSIM simulator [39]. The SMTSIM simulator has extensive support for both multithreaded and multi-core execution, and for this study it is configured for multi-core. In this study, speculative threads are executed on available cores on a CMP with a varying number of in-order execution cores. Table I gives the configuration details of the default architecture we simulate.

The simulator models spawning at instruction fetch, exposes wrong-path memory operations to coherence, and squashes threads in reaction to various memory coherence operations. It also models the effect of spawn points fetched on the wrong path (a result of the design decision to spawn threads when the spawn point is fetched rather than at commit) by ensuring that the target core is occupied and unavailable for execution until the branch mispredict is discovered.

We use the SPEC CPU2000 benchmarks, specifically the Alpha binaries for SimpleScalar [40]. Using reference inputs, we simulate the execution of one hundred million instructions starting at the standard SimPoint [41].

## V. Leveraging TM for SpMT

The two main requirements of any SpMT memory design are: (1) Speculative State Buffering – speculative state must be invisible to less speculative threads as well as protected from reaching lower levels of the memory hierarchy. (2) Conflict Detection – the hardware must be able to detect if a speculative thread has used an incorrect memory value or a hazard is present.

Hardware transactional memory supports each of these requirements. Speculative state is tagged as transactional in a transactional buffer or the cache, depending on the implementation. It is possible, in some implementations, for the same value to exist in multiple transactional caches, with different values. Memory data conflict detection is also supported and relies on modifications to the cache coherence protocol.

For SpMT, these features of TM can be leveraged to detect memory dependencies (and hazards) between threads. By starting a transaction when spawning a speculative thread, the TM hardware will protect all memory operations. Should a memory dependence be caught, the TM hardware can squash the offending speculative thread and discard its speculative state. Which memory dependences cause a speculative thread to be squashed depends on the TM implementation.

### A. The Baseline Transactional Memory Design

Our baseline HTM design is based on the original proposal by Herlihy and Moss [1] which has been adopted by more recent TM proposals including LTM [2], UTM [2], PTM [31], LogTM [32], and VTM [8]. This design has the same conflict detection semantics as those in LogTM-SE [9] and TokenTM [3]. It is also likely to be similar to the hardware support in SUN's Rock Processor [14], [33]. However, like most recent studies, we buffer speculative state in cache, whereas the original work used a special Transactional Buffer. In our implementation, a transactional bit is added to each cache line to differentiate those cache lines affected by the current transaction. A transactional read bit is added to identify lines read by the current transaction. A memory conflict occurs whenever the write set of one thread and the read or write set of another thread overlap. When a conflict occurs, the TM arbiter (which handles the restarting of threads) consults the thread ordering and squashes the more speculative thread. This TM design uses a write-invalidation protocol and any transactional, dirty cache lines are invalidated when squashed. We refer to this design as the Unordered Line Invalidate (ULI) design, to distinguish it from other designs we will introduce.

The baseline TM design buffers speculative state in the L1 data cache, with an 8-entry dedicated victim cache for transactional state overflow. In the rare event that the victim cache becomes full with transactional data for a speculative thread, we cause the thread to wait rather than invoking

software. Likewise, if the non-speculative thread overflows its victim cache, we squash all speculative threads. Although squashing is not required, this is not a common case in our studies, so the baseline HTM (without ordering) approach of squashing is used in that event.

Conflict detection is handled eagerly per memory operation by modifying the coherence protocol. Non-transactional coherence follows a traditional MESI protocol [42]. TCC [5] and Bulk [4] perform lazy conflict detection on a per thread basis, incorporating burst transfers of write set information. Lazy conflict detection will be addressed in Section VI-H1.

In this paper, we assume a system that has the ability to execute either SpMT code or standard parallel TM code. Thus, any optimization on the path from TM to SpMT that adds significant overhead but only provides benefit for SpMT execution would be questionable. Similarly, an optimization that improves SpMT performance at the expense of normal parallel performance would also be undesirable.

It is likely that a standard TM architecture would have support for transactions larger than supported by the base hardware TM mechanisms [2], [3], [8], [31], [32]. This is critical because standard TM code will fail to make forward progress if a transaction is too large. However, in our SpMT architecture, we can simply abort a transaction/thread that overflows the buffer space (the victim cache in this case). This is the solution we simulate, because fall-back support for larger transactions would likely be too slow for a SpMT solution. Thus, which of the proposed large transaction solutions might be supported by the actual system is unimportant for this work.

### B. Baseline TM SpMT Performance

Although the conventional TM architecture described in this section ensures correctness by catching inter-thread dependences, it does not provide encouraging performance results. Using two cores and averaged over the SPEC CPU2000 benchmarks, we achieve a 1.10 speedup over baseline execution. (Results per benchmark are provided in Figure 1.)

## VI. Accelerating SpMT

The previous section revealed that the SpMT performance of an existing HTM design with eager conflict detection is limited. The focus of the remainder of this work is identifying the shortcomings of that design and evaluating a set of hardware additions that eventually enable full support for speculative multithreaded execution. The key issues we address are support for ordered transactions, forwarding between transactions, addressing false sharing, and cold-cache effects.

This approach allows us to map out a path from HTM to SpMT that will allow us to fully exploit the hardware support for HTM we expect to see on future processors, while allowing the more aggressive speculative parallelization enabled by speculative multithreading.

### A. Ordered Transactions

Of the various HTM designs, TCC [5] and Bulk [4] support ordering between transactions [5]. In these designs, conflict detection is handled at transaction commit and commits can be forced to occur in an explicit order. The performance of these designs is in Section VI-H1. However, allowing for ordering between transactions in an eager conflict detection HTM, where detection occurs per memory operation, requires a very different implementation than TCC or Bulk.

Our baseline HTM architecture with minimal support for SpMT we call ULI (Unordered Line Invalidate) for reasons that will become clear. We extend this baseline to create an Ordered Line Invalidate (OLI) design. In this design, the coherence protocol is aware of the thread ordering and uses that knowledge to avoid unnecessary squashes. Then, for OLI, a memory conflict consists of any write-set overlap between threads as well as overlap between the read set of a more speculative thread and the write set of a less speculative thread. Overlap between the read set of a less speculative thread and a write set of a more speculative thread does not cause a squash. But this requires special handling of that line in the less speculative thread, because once the thread commits, the line is no longer valid within the execution context of a future thread that might occupy this core. The solution we employ is the same as that proposed by SVC [30]. We mark the less speculative line as *stale*. By stale we mean that the data is valid during this transaction but must be discarded when the transaction is completed. A stale line can be thought of as a delayed invalidate. Thus, a *stale* bit is added to all cache lines. A stale cache line is written back (if necessary) and invalidated when a thread commits. As with the ULI design, if a conflict occurs the more speculative thread is squashed and its transactional, dirty cache lines are invalidated.

### B. Data Forwarding for Ordered Transactions

We can further exploit the thread ordering information now that we have introduced it into our coherence mechanisms. A logical extension to the OLI design recognizes that if a more speculative thread requests a cache line held dirty by a less speculative thread, this should not result in a squash. Instead, the cache line can simply be forwarded between cores. Since the more speculative thread can only be committed if all less speculative threads are committed, this operation does not impact correctness. To address this scenario, we propose the Ordered Forwarding Line Invalidate (OFLI) design. By allowing for forwarding, we no longer need to squash when a more speculative thread reads a line held dirty by a less speculative thread. In addition to the information tracked in the OLI design, this new design requires a *forwarded* bit. The *forwarded* bit recognizes that a cache line is clean but has been forwarded

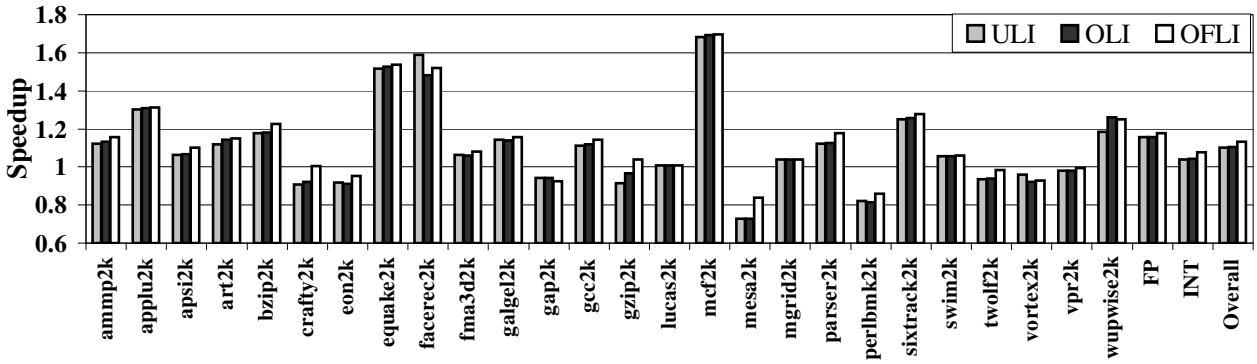


**Figure 1: Speedup per benchmark for ULI (unordered line invalidate), OLI (ordered line invalidate), and OFLI (ordered forwarding line invalidate) designs, relative to sequential execution.**

| Action | Holder | ULI | OLI | OFLI |
|---|---|---|---|---|
| R | R | | | |
| W | W | X | X | X |
| LSW | MSR | X | X | X |
| MSR | LSW | X | X | |
| MSW | LSR | X | | |
| LSR | MSW | X | | |

**Table II: Squashes caused by coherence actions for three memory designs. The more speculative thread is squashed in each case. The table indicates what happens when a single cache line is accessed by one core while the line is held by another core. LS = Less speculative. MS = More Speculative. R = Read. W = Write.**

from another core. Lines marked as forwarded need to be invalidated if this thread is squashed.

A summary of coherence actions that result in a squash can be found in Table II for the coherence designs discussed so far – ULI, OLI, and OFLI.

### C. Effectiveness of Line Granularity TM for SpMT

The speedup over the baseline (single thread execution) per benchmark for our first three memory designs is shown in Figure 1. In the majority of our benchmarks, the OFLI design outperforms the other designs.

Our intuition that adding ordering and forwarding between threads will result in more speculative threads committed is validated in that we see an average of 144k, 154k, and 156k threads committed in ULI, OLI, and OFLI respectively. Unfortunately, these improvements did not result in a significant performance gain (about 3% over ULI). In addition, our results are being offset by an overall loss in memory performance caused by cold cache effects. Using these memory designs for SpMT, the average memory access time has increased by 20% over single-threaded execution. As we successfully commit threads, the stream of committed execution flows from core to core, and we become vulnerable to cold caches. Section VI-H examines this phenomenon.

Also of note is that a number of the benchmarks suffer a slowdown. The majority of this slowdown can be attributed to SpMT overheads and memory slowdowns. For these benchmarks, a less aggressive spawn policy would be beneficial.

Averaged across all benchmarks, these results show that neither ordered transactions nor data forwarding have significantly improved the viability of speculative multithreading.

### D. Eliminating False Sharing

Prior SpMT work has shown that word-granularity speculative state information is necessary for strong performance [29], [30]. Our previously described HTM designs maintain state information at a cache-line granularity. This keeps coherence cost low, but results in a number of unnecessary squashes. In fact, for our OLI design, 72% of all squashes were due to false sharing. This number would be even higher, except that after threads repeatedly fail to commit due to false dependences, they stop being spawned.

There are two reasons tracking coherence at line granularity can cause excessive squashes. The first is that a read by a more speculative thread followed by a write of a different word in the same cache line by a less speculative thread need not result in a squash. However, false sharing would force a squash in our line-granularity designs. The second reason is the inability to reconstruct lines correctly in the presence of write sharing. When two threads write to the same cache line, a squash is required in earlier designs simply because it is impossible, without word-granularity tracking, to determine which word(s) from each dirty line should be preserved in the correct version. Each of these problems can be partially addressed by keeping a read and write bit per word per cache line. Writes of less than a word continue to be problematic, but they are infrequent in most programs – we deal with byte writes in Section VI-F.

Previous SpMT solutions, if they address this issue, require significant additional coherence traffic to resolve word-granularity versioning issues. Instead, we introduce a new solution by adding a *safe* bit to each line to eliminate unnecessary false squashes. When a more speculative thread has read a word, and a less speculative thread later writes a different word in the same line, our previous designs signal

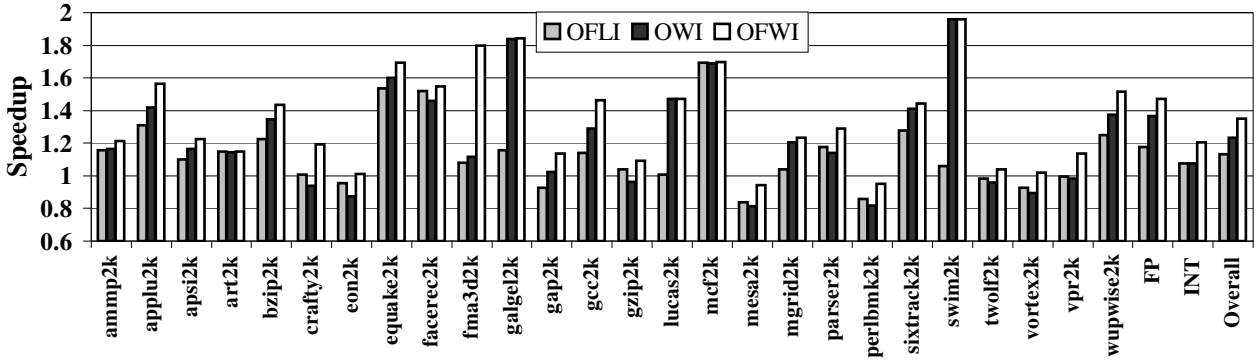


**Figure 2: Speedup per benchmark for OFLI (ordered forwarding line invalidate), OWI (ordered word invalidate), and OFWI (ordered forwarding word invalidate) designs, relative to sequential execution.**

a squash. Instead we mark the line as *unsafe* in the more speculative thread's cache. That thread can still read data it previously read or wrote, but accesses to any other word in an *unsafe* line require a coherence operation to get the latest version of the line. This is somewhat conservative, because we keep only a single *safe* bit, rather than one per word. Words written in unsafe lines are written back and invalidated when the thread is committed. On a squash an unsafe line is invalidated.

Without a safe (unsafe) bit to mark that only a portion of a line is valid, the coherence mechanism would either have to force the thread to squash (when we mark it unsafe) or immediately generate coherence traffic to collect all previously untouched words (which could be dispersed in various caches). The latter solution would likely cause prohibitive bus congestion. We evaluated the former solution and found that 55% of committed threads would have been squashed if it were not for the safe bit.

We add two new memory designs, then, which both support coherence tracking at the word level. They are the Ordered Word Invalidate (OWI) design and the Ordered Forwarding Word Invalidate (OFWI) design. They are analogous to the OLI and OFLI designs, respectively, but with word-level invalidation and a *safe* bit. In the OFWI design, we still require only a single *forwarded* bit per line.

### *E. Word Granularity Results*

The results for the word-granularity coherence tracking designs are shown in Figure 2. These results demonstrate that the ability to track coherence status at a word granularity has a significant impact on SpMT performance. For the OFWI design, for example, we now experience gains of 47% for floating point, 35% overall, and a maximum speedup of 96%. Also of interest is that word-level tracking suddenly makes data forwarding much more important. We see this because the difference between OFWI and OWI is much larger than the difference between OFLI and OLI seen previously. In fact, for SPECint, word-level tracking alone provides no performance gain, but in combination with forwarding the gains are significant.

The hardware overhead of word granularity is not insignificant, but it is clearly an important enabler of SpMT. But the advantage will not be limited to SpMT. Other parallel code using standard transactional memory will also benefit, in some cases likely significantly, from the elimination of transactional aborts due to false sharing [35]. Not only does eliminating false sharing in a TM system improve performance, but also insulates the programmer from yet another complexity of traditional parallel programming (false sharing).

### *F. Byte Granularity Results*

All word-granularity designs address byte granularity writes by maintaining a single bit per cache line to signify that a sub-word granularity write occurred to the line. When these writes make it impossible to reconstruct the word, the more speculative thread is squashed. The impact of byte hazard squashes is minor. We see 1% and 5% of thread squashes on average to be caused by byte hazards for OWI and OFWI, respectively. But the performance loss is virtually none. This would indicate that those threads that are lost due to byte hazards were typically either going to be squashed anyway, or were not going to be high quality speculative threads.

### *G. Word Granularity for Unordered Transactions*

One goal of this research is to identify a unified architecture for optimistic concurrency which effectively supports both TM and SpMT. Standard TM code will not always make use of all of the features we have proposed. The differences will be seen only when the code makes use of unordered transactions (which may be common in TM code).

In order to execute unordered transactions (transactions with equal ordering) written for traditional TM systems, minor modifications to the protocol above are required. For transactions of equal order, any cache line transactionally modified by another thread is made unsafe. Reads to previously untouched words in unsafe lines are filled by less speculative threads or by the L2 rather than by forwarding between equally ordered threads. Any overlap of the word

read-set and the word write-set between equally ordered transactions results in an abort.

### H. Write Update

Despite the performance improvements resulting from adding word granularity information to cache lines, memory performance remains a limiting factor. As the memory designs improve their support for completion of speculative threads, coherence misses and memory delay increases. The coherence misses increase from 39% of data cache misses for OFLI to 47% for OFWI. The slowdown in average memory delay over baseline execution also increases from 1.18 for OFLI to 1.40 for OFWI. There is a cold cache effect, and these numbers show that a significant factor is coherence misses.

With a write-invalidate cache coherence protocol, writes cause line invalidations which cause coherence misses. For example, on a four-core SpMT processor, a frequently-read variable will be in every cache. Each write to that variable will potentially result in three eventual coherence misses.

Our next design seeks to address the latter of these concerns by implementing a write-update protocol. Few cache-coherent multiprocessors have incorporated the write-update protocol [43], despite significant early research. This is because few data items are write-shared by many threads of a typical parallel application at once. However, with SpMT, especially in the absence of compiler support, this characteristic is not necessarily preserved. Thus, we investigate the utility of the write-update protocol in this architecture.

Before outlining the addition of the write-update protocol to our OFWI design, we will discuss our LAZY design which represents TCC [5] and Bulk [4]. The discussion regarding this design has been delayed because TCC and Bulk both have word-granularity conflict detection and a modified write-update protocol and are hence better compared against an eager-detection HTM with similar features.

*1) LAZY:* Two HTM proposals support lazy conflict detection – no information about modified lines are visible outside the core until a transaction commits. Both of these HTM proposals support word granularity. These proposals are represented by the design *LAZY* in our results. The first, TCC [5], [6], supports word-granularity coherence and the write-update protocol. TCC proposes replacing per-memory operation coherence with the bursting of write sets at commit. All execution remains completely isolated during a transaction. At commit, we impose an eight cycle arbitration delay and then broadcast all words in the transaction set (at two words per cycle). The second, Bulk [4] also supports word-granularity and performs write-update on lines with false-sharing word conflicts. It should be noted that lazy conflict detection can be highly advantageous in an environment where memory conflicts are infrequent. We do not expect that to be the case with SpMT, but we still model this approach to better understand the tradeoffs in a combined SpMT/HTM environment.

For the primary LAZY results, we will specifically model TCC; however, we also examined an idealized Bulk design and found that the difference between the two in this context is small. For SpMT on two cores, the only significant differences are that TCC keeps additional state per cache line whereas Bulk maintains signatures — Bulk may transmit less information when performing conflict detection, and Bulk is susceptible to false conflicts. For more than two cores, in addition to these differences, Bulk supports "Partial Overlap" in which data written by a speculative thread prior to spawning a new thread is made available to that new thread. It is important to note that the Partial Overlap optimization is not the same as forwarding. Forwarding allows for values to be transmitted between simultaneously active threads whereas Partial Overlap can only transfer values to new threads at spawn.

In the event of a transactional state overflow (the victim cache fills) by the non-speculative thread, we commit the transaction rather than squash.

In the context of our work, LAZY has a number of advantages as well as disadvantages. By committing the entire write set of a thread when the transaction commits, many coherence operations can be merged and wrong-path memory operations remain invisible to other threads. Additionally, since all information is maintained at a word level, words rather than cache lines are transmitted at transaction commit. However, since writes are only visible at cache commit, LAZY designs are unable to forward data from one cache to another. Finally, for TCC, the burst of a large number of writes consumes the buses and all data caches for a number of cycles.

In TCC, words written in a transaction are buffered. To solve sub-word granularity hazards, a single bit per word address in the store address FIFO is maintained to denote if a write at sub-word granularity occurred. Squashes are required if sub-word writes to the same word in different threads occur.

All data presented here assumes that transactions are committed either when the non-speculative thread overflows the victim cache (a rare event in our results) or when it commits (reaches and commits the CQIP). We explored the possibility of more frequently committing the non-speculative thread to expose the write set to the other threads earlier; however, this provided no real gain due to additional commit overhead and the still present, although reduced, delay in notifying threads of conflict violations and inability to forward data when accessed by another core.

*2) Ordered Forwarding Word Update (OFWU):* Adding write update to the protocol introduces a new hazard when we have more than two cores. Consider the case of three speculative threads, thread 0 being least speculative and thread 2 most. All share a cache line. Thread 0 writes a

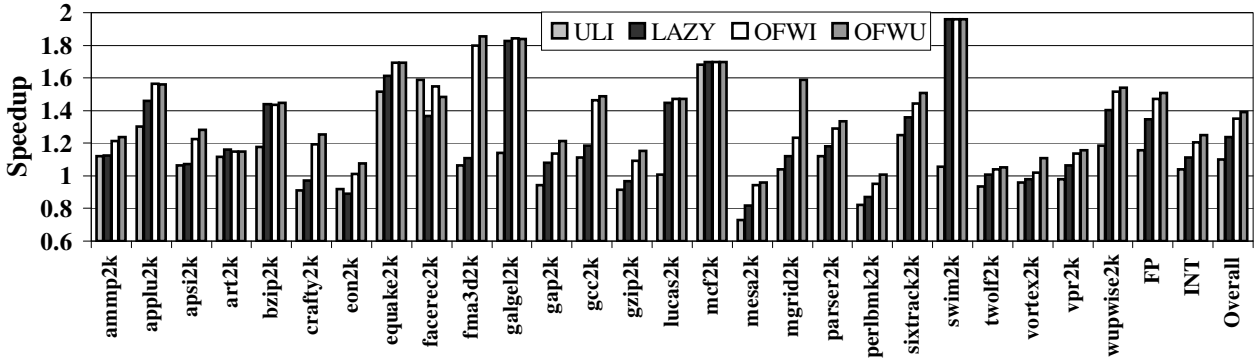


**Figure 3: Speedup per benchmark for ULI (unordered line invalidate), LAZY (lazy conflict detection), OFWI (ordered forwarding word invalidate, and OFWU (ordered forwarding word update) designs, relative to sequential execution.**

word of that line. Because of the write-update protocol, both threads 1 and 2 get the update but must mark the line as forwarded. Now thread 1 writes a different word in the line, and thread 2 again gets the update and is consistent. But if thread 0 now writes the same word as thread 1 wrote, thread 2 does not know what to do, because it does not store enough information (and it would be prohibitively expensive at a word granularity) to track from where it received each value. To address this situation at minimal cost, we add a *forward distance* field to each line. The observation is that as long as updates to the line happen in monotonically increasing order (from less speculative to more speculative) then there is no ambiguity. The *forward distance* allows us to ensure that the distance (in threads) from the last writer to this thread is decreasing. Once the order departs from that, the line is marked as unsafe by clearing the *safe* bit. When the line is unsafe, the local thread may read words that it has previously read or written, but reads of other words result in a coherence action to restore the full line to a correct state.

*Forward distance* allows for the preservation of updated cache lines and helps by reducing subsequent coherence misses due to invalidations of safe lines. With four cores, our OFWU design reduces memory slowdown by 20% compared with OFWI. Nearly half of this improvement is lost without forward distance bits. Since the goal of the update protocol is to improve memory performance, the *forward distance* is an inexpensive and effective tool for an implementation with more than two cores. Our performance with four cores is further examined in Section VII.

*3) Update Design Results:* The results for the ULI, OFWI, LAZY, and OFWU designs are shown in Figure 3. LAZY HTMs perform better than the line-granularity eager conflict detection HTMs. However, LAZY HTMs do not perform as well as word-granularity designs with forwarding for this SpMT architecture. This is not surprising, as this is not the execution model for which the lazy conflict detection systems were designed.

Our LAZY design is based on TCC. One key advantage of Bulk over TCC is the amount of data transmitted at commit. To determine if this transfer time was a significant limiter for LAZY, we ran TCC with near-instantaneous transfer times. The results were only slightly better with an average speedup of 1.26 rather than 1.23. This shows that the delay of transferring words (rather than signatures) is not the primary limitation of LAZY for SpMT.

The biggest disadvantage of the LAZY design in this context is the delayed notification of conflicts. A thread that reads data that has been written by a less speculative thread will not only be unable to acquire that data, but will not even find out it needs to squash until the earlier thread becomes non-speculative and commits. In many cases, that could be a long delay, especially if the writing transaction must wait to become non-speculative. In theory, the ability to forward could be added to a lazy conflict detection HTM, but it would represent a dramatic change to both the coherence protocol and the philosophy of lazy conflict detection.

Dependence Aware TM (DATM) [35] also offers word granularity with write-update. As mentioned before, the transactional guarantees by DATM are too weak for our SpMT framework. However, the inability for previous transactions to write to words written by later transactions has a significant effect. Just adding that restriction to our OFWU model caused a slowdown of 9%, resulting in performance similar to OWI and LAZY designs.

Shifting from invalidate to update (OFWU) provided slightly improved performance, but the gains were small. In addition to the overall increase in IPC, coherence misses are reduced from 47% to 37% and the memory slowdown decreased from 1.40 to 1.22 from OFWI to OFWU. We see that average memory access time was improved, partially as a result of decreased coherence misses. But the surprising result is the high rate of coherence misses that remain.

In a conventional write-update machine, all coherence misses would be eliminated. But there are many more types of coherence misses in this system (we use the simplest definition of a coherence miss – a tag hit to an invalid line). There are the delayed invalidates that are not seen until a thread commits, as well as all the invalidates that happen

when a thread is squashed. These results show that nearly 80% of the coherence misses were caused by actions that the write-update protocol did not address. Further optimizations to address cold cache problems are part of ongoing future work.

### I. Summary

As stated previously, the magnitude of the gains achieved is not the point of this study, and that magnitude would vary with specific assumptions in our architecture. Of greater interest is what optimizations are important to reaching the potential of speculative multithreading without adding significant complexity to hardware TM.

We find word granularity coherence tracking and forwarding are both critical for SpMT performance. Word granularity adds some complexity and a modest amount of hardware overhead, but it should accelerate both SpMT and TM execution.

Despite the high rate of coherence misses, the write update protocol provides modest gains. Given the complexity of supporting that protocol (including difficult corner cases that do not exist in conventional systems), and the fact that based on past research we do not expect write update to be particularly useful for non-SpMT parallel code, this optimization does not appear to be warranted.

Finally, lazy conflict detection proposals provide better performance than existing eager conflict detection proposals but have lower performance than our more advanced designs because of the delay in exposing the result of earlier writes to later computation.

## VII. Generality of Results

All of our results so far have examined a specific SpMT architecture. This section varies the details of that architecture to verify that our preceding conclusions still hold. Three dimensions of our architecture likely to change are the number of cores utilized, the register dependence handling mechanisms, and the complexity of the cores. Previous results were restricted to a small 2-core implementation and this section examines a larger (4-core) configuration. We have been simulating a very accurate (actually, perfect) register value predictor to handle the prediction of spawned thread live-ins – again, near-perfect prediction is not necessarily unattainable [17] (with a performance cost). However, we also examine a much more conservative register value predictor.

Our TM designs were all run with two and four cores. The results for execution on four cores averaged across all benchmarks is shown on the left side of Figure 4. We see that we get higher overall speedup with four cores (as much as 45%), but the overall gains have not scaled particularly well primarily because of the increase in cold cache effects, which are the result of two phenomena. The first is the expected increase in cold cache misses as we migrate

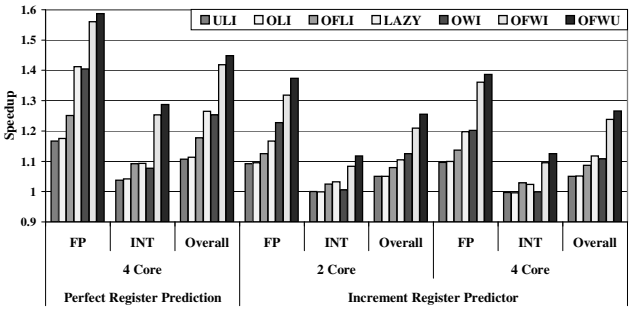


**Figure 4: Speedup for four cores with perfect register prediction and for two and four cores with an increment register predictor.**

between cores more frequently. The second phenomenon is that as we move to four cores, we successfully spawn more threads (fewer threads fail to spawn due to unavailable cores). The number of useful threads increases, but the number of squashed threads increases as well. Squashed threads typically result in invalidated lines. These effects are surprisingly large, resulting in an increase in average memory slowdown in the OFWU design from 1.22 with 2 cores to 1.44 with 4 cores. These effects should decrease with higher quality spawned threads (via compiler support, for example) and better solutions for the cold cache problem. Both are the focus of future research.

The more important result, however, is that despite significant changes in the execution details when running with four cores (more parallelism, more threads spawned, more coherence misses, etc.), all of the conclusions of our previous results remain. The word-granularity plus forwarding results provide over 40% performance gains, compared to the baseline TM design which still only gets about 10%. LAZY achieves a speedup of 26% which is less than the OFWI or OFWU designs. We continue to see that forwarding and word-granularity tracking must appear in concert to be effective. We also see the new result that while our aggressive designs provide some scaling with increasing number of cores, the default TM designs do not.

To evaluate the other end of the register prediction spectrum (from our perfect predictor), we have also modeled a basic 4K entry increment predictor [36]. Live-ins are tracked for spawnpoints using 2 bit saturating counters per register. Live-in registers are predicted at spawn by adding the increment indexed in the predictor to the current value held by the spawning thread. These predicted values are compared against the actual live-ins at thread commit and if the predictions are incorrect, the validating thread is squashed. This predictor both uses simple prediction, indexing, and update mechanisms and minimizes storage cost by only storing enough bits to capture most increment sizes. The only change to our previous methodology is that threads which perform illegal operations due to mispredictions are squashed.

The average results across all benchmarks are shown in Figure 4 for both 2 and 4 cores. With two cores, the baseline HTM design, ULI, continues to provide limited performance with a speedup of 1.05. LAZY performs better than ULI with a 1.10 speedup. Our OFWI and OFWU designs demonstrate significant improvements, 1.21 and 1.25 speedups respectively. The accuracy of the small increment predictor was quite high, between 63% and 76% depending on the memory design. The average number of live-in registers was between 2.5 and 3.0 depending on design. Undoubtedly, one factor in this result was the fact that threads with poor register prediction are eventually detected as undesirable spawns by our confidence counters.

We also evaluated the trends for out-of-order cores with 2 and 4 cores using either a perfect or conservative register predictor, and determined that the same trends and results hold with the more powerful cores. For example with the perfect predictor and four cores, we found speedups of 1.09 for ULI, 1.12 for OFLI, 1.17 for LAZY, 1.33 for OFWI and 1.35 for OFWU.

We see in all three cases (more cores, realistic prediction, and more complex cores) that all the important trends continue. We see similar gains over the baseline HTM design for the same best designs, LAZY falling between the baseline design and the better designs, word granularity being critical, and word-level granularity and forwarding providing highly synergistic gains.

## VIII. Conclusion

Speculative multithreading has the potential to significantly increase our ability to leverage highly parallel multicore, multithreaded processors for code that lacks the characteristics of traditional parallelism. Transactional Memory provides support for optimistic concurrency as well as an easier parallel programming interface and has such gained hardware support in processors proposed by industry. Prior work has recognized the ability for TM to support a domain of speculative multithreading limited to threads which are independent (dynamically).

This paper evaluates the path from TM to full SpMT. It evaluates a number of intermediate design points between TM and SpMT. It identifies a unified architecture capable of exploiting transactional parallelism, compiler-identified speculative parallelism, and hardware detected speculative parallelism. Three features are identified at being critical to SpMT performance - ordering of transactions, the ability to forward between transactions, and coherence tracking at word granularity. These trends persist across processors with in-order or out-of-order execution, two or four cores, and perfect or conservative register prediction.

## Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful insights. This research was supported in part by the Reserve Officers Association Henry J. Reilly Memorial Scholarship, NSF Grant CCF-0702349, and Semiconductor Research Corporation Grant 2005-HJ-1313.

## References

[1] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *20th International Symposium on Computer Architecture*, May 1993.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *11th International Symposium on High-Performance Computer Architecture*, Feb. 2005.

[3] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood, "TokenTM: Efficient execution of large transactions with hardware transactional memory," in *35th Annual International Symposium on Computer Architecture*, June 2008.

[4] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, "Bulk disambiguation of speculative threads in multiprocessors," in *33rd Annual International Symposium on Computer Architecture*, June 2006.

[5] 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 *31st International Symposium on Computer Architecture*, June 2004.

[6] A. McDonald, J. Chung, H. Chafi, C. C. Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun, "Characterization of TCC on chip-multiprocessors," in *14th International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2005.

[7] K. Moore, M. Hill, and D. Wood, "Thread-level transactional memory," in *Technical Report 1524, Computer Sciences Dept., UW-Madison*, 2005.

[8] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory," in *32nd Annual International Symposium on Computer Architecture*, June 2005.

[9] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. V. M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling hardware transactional memory from caches," in *13th International Symposium on High-Performance Computer Architecture*, Feb. 2007.

[10] T. Harris and K. Fraser, "Language support for lightweight transactions," in *18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, Oct. 2003.

[11] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer, "Software transactional memory for dynamic-sized data structures," in *22nd Annual Symposium on Principles of Distributed Computing*, July 2003.

[12] N. Shavit and D. Touitou, "Software transactional memory," in *Fourteenth Annual ACM symposium on Principles of distributed computing*, Aug. 1995.

[13] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun, "Programming with transactional coherence and consistency (TCC)," in *11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[14] M. Tremblay and S. Chaudhry, "A third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT SPARC®processor." in *IEEE International Solid-State Circuits Conference*, Feb. 2008.

[15] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *22nd Annual International Symposium on Computer Architecture*, June 1995.

[16] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun, "The Stanford Hydra CMP," *IEEE MICRO*, vol. 20, no. 2, Mar. 2000.

[17] C. Madriles, C. García Quiñones, J. Sánchez, P. Marcuello, A. Gonzàlez, D. M. Tullsen, H. Wang, and J. P. Shen, "Mitosis: A speculative multithreaded processor based on precomputation slices," in *IEEE Transactions on Parallel and Distributed Systems*, July 2008.

[18] M. K. Prabhu and K. Olukotun, "Exposing speculative thread parallelism in SPEC2000," in *10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2005.

[19] J. Steffan and T. Mowry, "The potential for using thread-level data speculation to facilitate automatic parallelization," in *4th International Symposium on High-Performance Computer Architecture*, Jan. 1998.

[20] V. Krishnan and J. Torrellas, "Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor," in *12th International Conference on Supercomputing*, July 1998.

[21] G. S. Sohi and A. Roth, "Speculative multithreaded processors," *Computer*, vol. 34, no. 4, 2001.

[22] P. Marcuello, A. González, and J. Tubella, "Speculative multithreaded processors," in *12th International Conference on Supercomputing*, Nov. 1998.

[23] H. Akkary and M. Driscoll, "A dynamic multithreading processor," in *31st International Symposium on Microarchitecture*, Sep. 1998.

[24] S. T. Srinivasan, H. Akkary, T. Holman, and K. Lai, "A minimal dual-core speculative multi-threading architecture," in *IEEE International Conference on Computer Design*, Oct. 2004.

[25] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," in *IEEE Computer*, July 2008.

[26] J. Chung, H. Chafi, C. Minh, A. McDonald, B. Carlstrom, C. Kozyrakis, and K. Olukotun, "The common case transactional behavior of multithreaded programs," in *Sixth International Symposium on High-Performance Computer Architecture*, Feb. 2006.

[27] C. von Praun, L. Ceze, and C. Cascaval, "Implicit parallelism with ordered transactions," in *12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Sep. 2007.

[28] M. Franklin and G. S. Sohi, "ARB: A hardware mechanism for dynamic reordering of memory references," *IEEE Transactions on Computers*, vol. 45, no. 5, May 1996.

[29] M. Cintra, J. F. Martńez, and J. Torrellas, "Architectural support for scalable speculative parallelization in shared-memory multiprocessors," in *27th Annual International Symposium on Computer Architecture*, June 2000.

[30] T. Vijaykumar, S. Gopal, J. Smith, and G. Sohi, "Speculative versioning cache," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 12, Dec. 2001.

[31] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, B. Calder, and O. Colavin, "Unbounded page-based transactional memory," in *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2006.

[32] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood, "LogTM: Log-based transactional memory," in *12th International Symposium on High-Performance Computer Architecture*, Feb. 2006.

[33] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," in *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[34] A. Shriraman, S. Dwarkadas, and M. L. Scott, "Flexible decoupled transactional memory support," in *35th Annual International Symposium on Computer Architecture*, June 2008.

[35] H. E. Ramadan, C. J. Rossbach, and E. Witchel, "Dependence-aware transactional memory for increased concurrency," in *41st Annual ACM/IEEE international symposium on Microarchitecture*, Nov. 2008.

[36] P. Marcuello and A. Gonzélez, "Thread-spawning schemes for speculative multithreading," in *Second International Symposium on High-Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, Feb. 2002, p. 55.

[37] T. N. Vijaykumar, "Compiling for the multiscalar architecture," in *Ph.D. Thesis. University of Wisconsin-Madison*, 1998.

[38] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry, "Compiler optimization of scalar value communication between speculative threads," in *SIGPLAN Notices*, 2002.

[39] D. Tullsen, "Simulation and modeling of a simultaneous multithreading processor," in *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[40] T. Austin, E. Larson, and D. Ernst, "Simplescalar: an infrastructure for computer system modeling," in *Computer Vol. 35, Issue. 2*, Feb. 2002.

[41] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[42] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *11th Annual International Symposium on Computer Architecture*, Jan. 1984.

[43] E. McCreight, "The dragon computer system," in *Proceedings of the NATO Advanced Science Institute on Microarchitecture of VLSI Computers*, 1985.