# Optimizing Software Runtime Systems for Speculative Parallelization

PARASKEVAS YIAPANIS, DEMIAN ROSAS-HAM, GAVIN BROWN, and MIKEL LUJÁN,
University of Manchester

Thread-Level Speculation (TLS) overcomes limitations intrinsic with conservative compile-time auto-parallelizing tools by extracting parallel threads optimistically and only ensuring absence of data dependence violations at runtime.

A significant barrier for adopting TLS (implemented in software) is the overheads associated with maintaining speculative state. Based on previous TLS limit studies, we observe that on future multicore systems we will likely have more cores idle than those which traditional TLS would be able to harness. This implies that a TLS system should focus on optimizing for small number of cores and find efficient ways to take advantage of the idle cores. Furthermore, research on optimistic systems has covered two important implementation design points: eager vs. lazy version management. With this knowledge, we propose new simple and effective techniques to reduce the execution time overheads for both of these design points.

This article describes a novel compact version management data structure optimized for space overhead when using a small number of TLS threads. Furthermore, we describe two novel software runtime parallelization systems that utilize this compact data structure. The first software TLS system, `MiniTLS`, relies on eager memory data management (in-place updates) and, thus, when a misspeculation occurs a rollback process is required. `MiniTLS` takes advantage of the novel compact version management representation to parallelize the rollback process and is able to recover from misspeculation faster than existing software eager TLS systems.

The second one, `Lector` (Lazy inspECTOR) is based on lazy version management. Since we have idle cores, the question is whether we can create "helper" tasks to determine whether speculation is actually needed without stopping or damaging the speculative execution. In `Lector`, for each conventional TLS thread running speculatively with lazy version management, there is associated with it a lightweight *inspector*. The inspector threads execute alongside to verify quickly whether data dependencies will occur. Inspector threads are generated by standard techniques for inspector/executor parallelization.

We have applied both TLS systems to seven Java sequential benchmarks, including three benchmarks from `SPECjvm2008`. Two out of the seven benchmarks exhibit misspeculations. `MiniTLS` experiments report average speedups of 1.8x for 4 threads increasing close to 7x speedups with 32 threads. Facilitated by our novel compact representation, `MiniTLS` reduces the space overhead over state-of-the-art software TLS systems between 96% on 2 threads and 40% on 32 threads. The experiments for `Lector`, report average speedups of 1.7x for 2 threads (that is 1 TLS + 1 Inspector threads) increasing close to 8.2x speedups with 32 threads (16+16 threads). Compared to a well established software TLS baseline, `Lector` performs on average 1.7x faster for 32 threads and in no case ($x$ TLS $+x$ Inspector threads) `Lector` delivers worse performance than the baseline TLS with the equivalent number of TLS threads (i.e. $x$ TLS threads) nor doubling the equivalent number of TLS threads (i.e., $x + x$ TLS threads).

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.4 [**Programming Languages**]: Processors—*run-time environments*



This research was conducted with support from the UK Engineering and Physical Sciences Research Council, on grants EP/G000662/1 and EP/F023855/1. Dr. Luján was supported by Royal Society University Research Fellowship.

Authors' address: The University of Manchester, Oxford Road, Manchester, M13 9PL, UK. Correspondence email: pyiapa@gmail.com.


DOI 10.1145/2400682.2400698 http://doi.acm.org/10.1145/2400682.2400698

General Terms: Languages, Design, Algorithms, Performance

Additional Key Words and Phrases: Thread-level speculation, speculative parallelization, runtime parallelization, multicore processors, loop-level parallelism, inspector threads, memory overhead, SPECjvm2008

**ACM Reference Format:**
Yiapanis, P., Rosas-Ham, D., Brown, G., and Luján, M. 2013. Optimizing software runtime systems for speculative parallelization. ACM Trans. Architec. Code Optim. 9, 4, Article 39 (January 2013), 27 pages.
DOI = 10.1145/2400682.2400698 http://doi.acm.org/10.1145/2400682.2400698

## 1. INTRODUCTION

Parallelizing applications is the prevailing answer to the ubiquitous presence of multicores. Current practices of parallel programming remains a difficult task for most programmers and parallelizing compilers are over-conservative, especially with applications that use pointers and dynamic data structures. In many cases, loops offer hidden parallelism that a static parallelizing compiler cannot prove until the application runs. A solution to this problem, namely *Thread-Level Speculation* (TLS), also known as *Speculative Multithreading* (SpMT), is to speculate at runtime.

In a general TLS system, the sequential program is optimistically transformed into a parallel one, assuming that the parallel execution will maintain the sequential semantics of the original program; that is, no data dependencies. While the application executes, each parallel running thread collects and maintains information regarding all of its memory accesses. For example, any updates (stores) from a given thread are kept locally to that thread, instead of written immediately back to memory, until proven to be correct. Since those threads have not proven to be successful while still executing, they are called *speculative threads*. During, or at the end of speculative execution, an inspection phase takes place to ensure that there were no data dependence violations of the sequential semantics of the application. If a thread did not conflict with another, then it is safe to propagate its modifications back to memory, an action which is called *commit* in this context. Otherwise, the offending thread has to *squash*, that is, discard any temporary (local) modifications and re-execute its code. When threads squash, a procedure initiates to ensure that those threads will undo their modifications properly and leave the memory state as it was before the squash occurred. This procedure is called *rollback*.

Previous work on TLS looked at hardware implementations [Cintra et al. 2000; Chen and Olukotun 2003; Prabhu and Olukotun 2005; Johnson et al. 2007; Luo et al. 2009; Madriles et al. 2009; Kim et al. 2010] as well as in software [Dang et al. 2002; Cintra and Llanos 2005; Oancea et al. 2009; Mehrara et al. 2009; Chen et al. 2008; Tian et al. 2010; Raman et al. 2010; Kim et al. 2012], just to name a few. However, no widely available architectures or compilers have incorporated TLS. Nonetheless, the published limit studies suggest that it could be profitable to use TLS [Packirisamy et al. 2009; Ioannou et al. 2010], although we should focus on optimizing for small number of threads. Motivated by this potential, we have conducted experiments to verify whether this applies in a real machine, besides limit studies.

At a high level, there are two main design approaches (explained in the next section) that have been followed by the software TLS literature, in terms of how a system maintains its speculative state. We present two novel TLS systems, one for each of these design direction. The contributions are the following.

—We provide a compact data structure to represent the dependency tracking for a TLS system. We show space overhead reduction of 5x on average compared to state-of-art approaches.
—We present `MiniTLS`, a software TLS system for Java applications, that relies on eager memory data management. Speculative threads modify directly data which needs

to be rollback when misspeculation occurs. This eager treatment provides faster execution in the absence of data dependencies.
—MiniTLS outperforms state-of-the-art software TLS systems and it is the only one which parallelizes the rollback process under Eager Version Management.
—We present a new software TLS system, Lector using lazy version management. Compared with a state-of-the-art lazy TLS, Lector shows performance improvements on average of 1.7 times faster than the baseline.
—We provide a novel algorithm for accelerating TLS systems applicable to any type of implementation. We show that applying this technique, improves speedup on average 1.7 times for 2 threads increasing close to 8.2 times speedups with 32 threads.

The rest of this article is organized as follows: Section 2 provides background on TLS (Section 2.1) and the main design approaches when implementing software TLS (Section 2.2). Section 3 describes our first system, MiniTLS, as well as our novel parallel rollback algorithm. We illustrate how we take advantage of the information provided by TLS limit studies to optimize MiniTLS. Oancea et al. [2009] introduced a top performing software TLS work using eager memory management. Their contribution was how to eliminate any associated synchronization when accessing the data structures holding the dependency information. Oancea et al. [2009] traded off increasing the size of the eager memory management data structures to remove synchronization and optimize performance. Hereafter, we refer to the software TLS developed by Oancea et al. as SpLIP. In our experiments, we compare directly the performance delivered by MiniTLS vs. SpLIP. Section 4 describes our second system, Lector as well as our novel idea of using inspector threads in the role of helper threads. Section 5 reports the speedup results as well as speculative operations and memory overhead comparisons using seven Java benchmarks, with three belonging to SPECjvm2008. Note that two of the seven benchmarks do include data-dependencies. Section 6 presents related work and Section 7 summarizes this article.

## 2. BACKGROUND

### 2.1. Thread-Level Speculation

Assume, that we wish to parallelize the loop shown in Figure 1(a). Similarly, assume that the array indexes cannot be resolved until runtime. That is, there is no feasible means of performing any static analysis (manual or automatic) to prove correct parallel executions by eliminating the possibility of data dependencies across threads. This can be the case for example, where $i$ or $j$ are the result of accessing an indirection array. Therefore, standard parallelizing compilers must conservatively produce sequential code for the loop in order to guarantee correct execution. Consider now the instance of sequential execution shown in Figure 1(b). Clearly, the values populated for the array indexes did not yield any data dependencies amongst them, and thus, the compiler could have generated a parallel code such as the one in Figure 1(c) and allow the application to enjoy the performance profits.

*Thread-level Speculation* (TLS) circumvents this problem by executing the threads (which are formed by loop iterations in this case) in parallel assuming that the run-time values of $i$ and $j$ will not trigger any cross-thread conflicts. In this case, TLS would execute the loop iterations in parallel while at the same time underlying mechanisms would monitor every speculative access to ensure that the parallel execution will produce the same results as if the program was executed sequentially. In addition, any memory updates are buffered locally to the thread rather than written-back to main memory. Figure 2(a) shows the case where all speculative threads executed successfully and thus are allowed to *retire* or *commit* by propagating the buffered updates back to main memory. Sometimes we have the case of a memory dependency like the

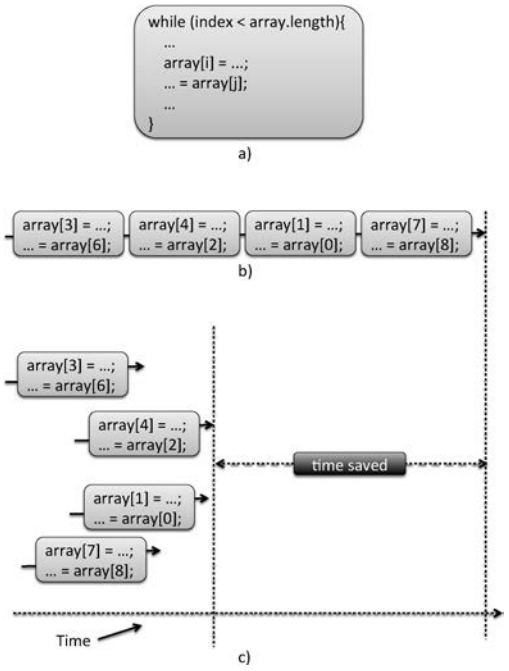


Fig. 1. (a) Code fragment of loop to be parallelized. (b) Sequential execution. (c) Sample parallel execution.

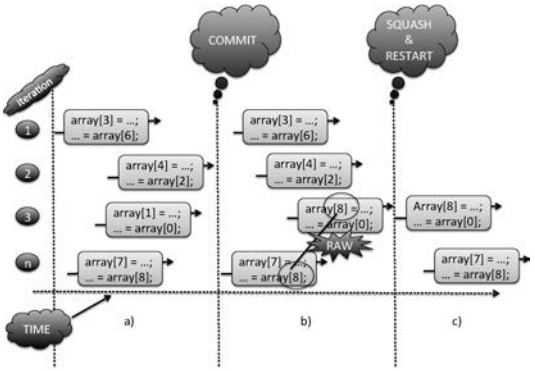


Fig. 2. (a) Speculative loop execution without dependencies. (b) Speculative loop execution with dependency. (c) Re-execution of offending threads.

one shown in Figure 2(b). In this case, a thread (iteration) has loaded a value that was not produced by the correct store. This action causes a *Read-After-Write* (RAW) data dependency. As a result, the offending threads need to *squash* by initiating the *rollback* procedure (in this case, discard any buffered updates), and re-execute in the correct order (see Figure 2(c)).

Implementing the underlying mechanisms that will guarantee correct execution in TLS require certain design decisions. Three important design characteristics in a TLS system are: *concurrency control*, *version management*, and *conflict detection*. The next paragraphs elaborate more on those three.

### 2.2. Design Choices

There are various axis of implementation options when designing a TLS system. Although, various studies have tried to evaluate different designs, there is no solid experimental study on which choice is better than another in the absence of the application's behavior. In other words, design choice depends greatly on the target applications.

*2.2.1. Concurrency Control.* *Concurrency Control* refers to the way a system detects and resolves a conflict when that occurs. There are two general categories of concurrency control: *Pessimistic Concurrency Control* (PCC) and *Optimistic Concurrency Control* (OCC). In PCC, a speculative thread acquires exclusive ownership of the location to be accessed. This is done by using some form of locking primitive. While the thread owns the location, the underlying system checks for conflicts. In the unfortunate scenario that a conflict occurs, it will be detected and resolved (depending on the system's design) immediately. With OCC, multiple speculative threads are allowed to access the same location simultaneously while conflict detection and resolution can happen at a later stage,

anytime before the commit phase takes place. OCC is normally more appropriate when conflicts are expected to be rare, otherwise it may be producing significant wasted work.

*2.2.2. Version Management.* While threads execute speculative code, different versions of data are being produced. *Version Management* refers to the way those different versions are maintained by the TLS system. Typically, there are two major approaches for that: *Lazy Version Management* (LVM), also known as *deferred updates* and *Eager Version Management* (EVM), also known as *in-place updates*. When LVM is used as a choice, speculative threads require at least a buffer per thread in order to keep any tentative stores. Speculative loads search first the thread's local buffer in case they find an associated value for that location there. If not, the value needs to be loaded from memory. Speculative stores just need to add or update the corresponding value in the local buffer. At the end of a thread's speculative execution and provided that there was no conflict for this thread, the results from the local buffer are propagated to main memory to make visible the updates to other threads. In the case of a conflict, the speculative thread only needs to discard its local buffer, since there was no modification of the actual data in memory.

EVM systems, on the other hand, update memory locations directly when the speculative store occurs rather than delaying the action. This involves having a buffer that preserves the original value of the memory location just before the update. This buffer is known in the literature as the *undo log* since in the case of a conflict the log is used to restore the memory back to a correct state. Speculative loads can use the values from memory as the new values are already there. Upon successful commit, the thread simply discards the undo log without requiring any value propagation as in LVM.

*2.2.3. Conflict Detection.* A conflict can occur when two or more speculative threads access the same memory location in a way that cause a data dependency violation. Depending on the version management system used, different actions may or may not cause violations. There are three type of data dependencies: *flow*, *anti*, and *output dependencies* which give rise to Read-After-Write (RAW), *Write-After-Read* (WAR), and *Write-After-Write* (WAW) hazards respectively. A RAW violation is caused when a thread loads a value that was not produced by itself. WAR and WAW violations arise due to reuse of memory locations. A system that used LVM does not need to worry about WAR and WAW dependence violations since the updates are buffered and speculative loads use those instead. This is somewhat similar to the *Register Renaming* action taken at the hardware level to prevent those kind of hazards. In contrast, EVM has to take precautions for WAR and WAW dependence violations since the values in memory are always up-to-date. Nevertheless, both EVM as well as LVM systems need to be observant for RAW violations.

There are two types of conflict detection: *Lazy Conflict Detection* (LCD) and *Eager Conflict Detection* (ECD). LCD implies that threads may be allowed to run through their respective speculative code without checking for conflicts on every access. Conflict detection can occur at a later stage as long as that happens before thread commit. In this way eager checks during execution are eliminated. ECD checks for conflicts usually on every speculative access in order to catch any violations as soon as they arise. The idea here is to prevent any wasted work after a conflict has happened.

*2.2.4. Scheduling.* Since the loop is parallelized automatically from sequential code, the execution behavior is rather unpredictable. The way iterations are scheduled to run across the available threads can have significant impact in the final performance. Traditional scheduling possibilities include *static* and *dynamic scheduling*. Static scheduling partitions the loop into equal chunks of iterations based on the number of available threads. The thread that will execute a particular chunk is decided statically. In

contrast, dynamic scheduling allows those chunks to be assigned to threads at runtime. Both of these are not very well suited for TLS since they can cause, load imbalance, increase the probability of dependence violation, and increase memory overhead. A different scheduling technique known as *Sliding Window* was studied by Cintra and Llanos [2005] and found to be a good alternative for TLS. Under sliding window, chunks of iterations are assigned into windows of size $W$. The window moves forward (slides) when all iterations in the window have finished. This allows better load balancing, decrease in likelihood of dependence violation, and better decoupling of memory overhead.

### 2.3. Other Implementation Details

The main overhead in software TLS arises from maintaining the speculative state. Certain hardware solutions that apply LVM can overcome this problem using their L1 cache, which is already private to the processor and thus having the marking of speculative loads and stores essentially for free. Software approaches, on the other hand, rely on additional data structures to maintain marking information. Thus each potential speculative load or store will produce at least an extra load and store from the hardware point of view (to read and insert that item to the data structure). These data structures are normally accessed via monitors or *Compare-And-Swap* (CAS) operations to avoid data races between different threads.

Apart from the main design choices, many alternative implementations are possible. There is no solid study that shows that a certain design choice is better than another. It all depends mainly in the behavior of the specific running application. Generally speaking, a system that utilizes EVM might be more complicated to implement than one using LVM. Since stores are written in-place (eagerly), the designer has to also consider *Write-After-Read* (WAR) and *Write-After-Write* (WAW) dependencies, apart from *Read-After-Write* (RAW) dependencies.

## 3. MINITLS: SYSTEM DESCRIPTION

`MiniTLS` is based on Eager Version Management and we report its implementation using the Java programming language. This section also describes, for the first time in this article, the compact data structure used in our implementations.

### 3.1. General Idea

The focus of `MiniTLS` is loop parallelization. Loop iterations run in parallel while threads are monitored during execution for potential violations. Every memory location is protected by a lock primitive and as such only one thread at a time is allowed to have access on it. Threads update memory locations in-place and therefore any load performed by others is guaranteed to have the most up-to-date value. Before a thread updates a memory location, the original value is saved in a log, in case it is required by a rollback in the future. Marking and monitoring is facilitated by a shadow data structure (explained in the next section). Discovering a violation by a thread entails recovering memory to the latest known correct state by restoring values using logs from the offending thread as well as any other thread involved in the violation. Parallel execution restarts and continues in the same fashion until all loop iterations complete.

### 3.2. Data Structures

*Shadow Data Structure.* There is a shared data structure, called *shadow data structure* (see Figure 3), that maps every user-accessed address into an array of integers using a hash function. Figure 3 illustrates the case of sixteen running threads. This structure is an array of Java integer values. In this case of sixteen threads, each address in the user data space is represented by two consecutive 32-bit integer memory locations in the shadow array. The first location (named “mark”) is used to mark the

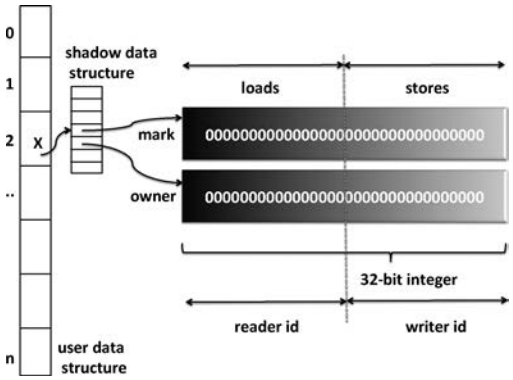


Fig. 3. Shadow data structure in `MiniTLS`.

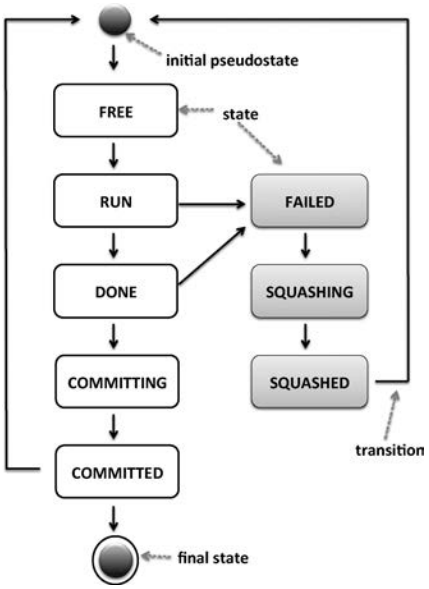


Fig. 4. Speculative thread lifecycle for `MiniTLS`.

thread(s) that performed a *load* and/or a *store* on that user address, and the second (named "owner") is used to indicate the thread(s) that are currently operating in the user address. The thread that currently performs a read or a write has to lock the address by setting the appropriate bit in "owner" in order to claim exclusive ownership to the location. Besides the action to be performed, the appropriate bit in "mark" is set before ownership is released.

For a thread $T$ that needs to access memory location $x$, the order of operations is as follows: First, $T$ checks that $x$ is available by issuing a CAS operation on $h = hash(x)$. $T$ indicates that is the owner of $x$ by setting the appropriate bit in "owner". Then, $T$ operates on $h$ (load or store) accordingly by setting the appropriate bit in "mark" to indicate the action performed. Finally, $T$ resets the bit in "owner" and releases the lock so another thread can access $h$ if needed. Note that, the bit location in the bit sequence acts as thread id and indicates the order of speculation. That is, a less significant bit indicates a less speculative thread.

The locking mechanism is totally flexible to the designer's choice. Figure 3 illustrates how the "owner" could be used for read-write locks that allow multiple readers but only one writer per user address at a time. In our implementation we did not find this added complexity worthwhile so the same lock is used for readers as well as for writers and thus requiring only half the space of the "owner". Our underlying locking mechanisms are bounded spin-locks. A thread may busy-wait (a finite number of times) for another to finish any work in the same location without blocking.

Assuming an 8-thread configuration, each location in the shadow structure requires 24 bits (although extended to 32 bits to avoid misalignments) to keep the owner-(8-bits)-reader-(8-bits)-writer-(8-bits) information. After the hash function determines a location in the shadow structure, the appropriate bits must be examined and updated. If thread 4 requires to read location $x$, then $hash(x)$ will be accessed in the shadow structure, and the information will be read and updated using a CAS operation. Assuming the contents of $hash(x)$ are empty $00000000_{hex}$, thread 4 will check no other thread is operating there; that is, the owner part is empty (i.e., $0000_{hex}$). Thus, the CAS

operation will succeed and set the owner part to $0100_{hex}$ as well as the reader part to $0100_{hex}$. This will leave the contents of $hash(x)$ being $0100 - 0100 - 0000 - 0000_{hex}$.

So, in essence, information for all the readers and writers of a particular user address, can be stored from as little as 6-bits for 2 threads to 96-bits for 32 threads. This solution represents a more compact way of memory representation compared to other solutions in the literature. We choose to experiment with only up to 32 threads and not more, motivated by the results in previous TLS limit studies (such as Ioannou et al. [2010]) which indicate there is no significant benefit with higher number of threads. Of course this solution can be easily extended to support more threads.

*Data Structures Private to Each Thread.* In addition to the shadow data structure, each thread has local read- and write-set implemented using array structures. The read set consists of a set of indexes accessed in the shadow array. This is used when the thread finishes, in order to reset all its accesses in the shadow array and avoid any potential false conflicts in later accesses. The write set, apart from the indexes accessed, it also keeps a record of the values that are updated in main memory. Since, this is an implementation of an in-place (eager) system, the memory values need to be recorded before being modified by a speculative thread. The write-set is also called an *undo log* in this context, since in case of a conflict it is used to undo all the speculative operations. Other in-place implementations [Oancea et al. 2009] require a time-stamp, for each memory location, to be recorded in the undo log. This provides a sense of order between multiple writer threads in case of a rollback. When multiple threads have accessed a location that has to be reverted, only the value of the thread with the earliest timestamp is used so that the program's sequential order is maintained. A proof found in Oancea et al. [2009] establishes the soundness of this idea. Unlike others, our implementation does not need this extra time-stamp. Since the compact shadow structure contains already all the writer threads of a given location, the one with the lowest ID (lowest bit) can be used to revert the value.

### 3.3. Speculative Operations

In TLS, threads are normally organized in terms of speculation order. For instance, in loop-level speculation, where threads execute different loop iterations in parallel, the thread that executes the first chunk of iterations is known as the *least speculative thread*. A thread is always more speculative than the thread that executes the previous set of iterations. Consequently, the thread that executes the last set of iterations is the *most speculative thread*. Such an order is useful in TLS, as it facilitates commit and rollback decisions in order to preserve the program's sequential order. Less speculative threads have the right to "kill" more speculative threads.

*Loads.* A speculative load operation by a thread $T$ to a location $x$ simply needs to set its corresponding bit in the shadow array. The bit is not set again if $x$ was already accessed by the same speculative slice. Before this is done, the thread needs to check whether a more speculative thread has performed a store operation in $x$, as this can cause a *Write-After-Read (WAR)* conflict. The code is shown in Figure 5.

*Stores.* A store operation needs to check whether a more speculative thread has performed a store in the same location to prevent *Write-After-Write* (WAW) conflicts. Similarly it also needs to check whether a more speculative thread has performed a load in the same location. This causes a *Read-After-Write* (RAW) conflict. If none of those conflicts occurs, then the thread can safely record the current memory value in its write set and perform an in-place update to the to-be-modified location (code not shown). Similarly to loads, the store bit is not set if it was already set earlier by the same thread. The code is shown in Figure 6.

```
specLoad(int hash, int threadID){

   // lock by setting "owner" bit
   // if cannot lock → throws Exception
   shadowStruct.lock(hash);

   int loads = shadowStruct.getLoads(hash);
   int stores = shadowStruct.getStores(hash);

   // (stores == 0) → no other previous (thread) stores
   // (stores <= threads) → no more speculative stores
   if( (stores == 0) || (stores <= threadID) ){

      Value v = memory.loadValue(hash);
      thread.setCurrentLoad(v);
      shadowStruct.setLoads(hash, threadID);
      thread.recordLoad(hash, threadID); //in read set
      shadowStruct.unlock(hash);
      return;

   }else { // (store > threadID) → a more speculative
           //                      store found
      thread.squash(); // WAR
   }
}
```

Fig. 5. Speculative load.

```
specStore(int hash, int threadID, Value v){

   // lock by setting "owner" bit
   // if cannot lock → throws Exception
   shadowStruct.lock(hash);

   int loads = shadowStruct.getLoads(hash);
   int stores = shadowStruct.getStores(hash);

   if( (stores == 0) || (stores <= threadID) ){

      if( (loads == 0) || (loads <= threadID) ){

         Value old_v = memory.loadValue(hash);
         memory.writeBack(hash, v);
         shadowStruct.setStores(hash, threadID);
         thread.recordStore(hash, threadID, old_v);
         shadowStruct.unlock(hash);
         return;

      }else{ //more speculative load found
         thread.squash(); // RAW
      }
   }else{ //more speculative store found
      thread.squash(); // WAW
   }
}
```

Fig. 6. Speculative store.

### 3.4. Conflict Detection

MiniTLS implements eager conflict detection, that is, conflicts are detected as soon as they occur. Another option would be to wait until the end of a speculative thread's execution before any conflict is detected. However, we found that checking for conflicts on every speculative operation is less costly, compared to the wasted execution when the detection is delayed. Furthermore, our system employs immediate conflict resolution. That is, a thread that detects a conflict pauses instantly any speculative execution, and initiates a rollback by notifying all the more speculative threads than itself, to squash. Since each speculative thread checks for conflicts on every speculative access, the action will take place immediately. All the more speculative threads, than the offending thread (including the offending one), will wait until the rollback process begins. As soon as all the less speculative (than the offending one) threads finish execution and the offending thread becomes the least speculative thread, rollback is ready to begin.

### 3.5. Scheduling Policy and Ordering

For scheduling the speculative threads, we have used a *sliding window* mechanism [Cintra and Llanos 2005]. In a sliding window policy, the number of active threads depends on the size of the window. There are two reasons why we have chosen this mechanism. First, due to the nature of the policy (chunks of iterations are scheduled in windows), the probability for data dependency violation is decreased and load imbalance is reduced. Second, it was found as a beneficial scheduling choice in previous experiments [Cintra and Llanos 2005]. MiniTLS uses a conservative sliding window implementation where the window is reloaded when all threads currently occupying the window have completed.

The mapping of iterations to threads within a window is done allocating contiguous sets of iterations of equal size. The mapping allows for the thread id to inform of how speculative each thread is; thread $T_0$ is always less speculative than $T_2$ and so on. We are doing a static block scheduling of the iteration space within a speculation window. Once all the threads within a window have completed, the shadow data structure is reinitialized and a new mapping of the following iterations is performed.

### 3.6. Rollback and Recovery

`MiniTLS` is the first software TLS eager management system to propose and implement a parallel rollback operation. This can help in reducing the overhead when misspeculation occurs. Those speculative threads that need to be squashed will take part in the rollback operation while those less speculative threads for which no data dependency has been found will wait for the rollback operation to finish. The rollback mechanism is started by `thread.squash()` in Figures 5 and 6. First, we need to identify which is the least speculative thread that modified each location. We can do this in parallel by allowing each participating speculative thread to visit its write-set data structure, and for each element in the write-set check in the shadow data structure whether any other thread has modified it. To access the shadow data structure, threads use CAS operations. Should more than one thread have written a given location, it is simple to know whether that thread is a least speculative thread to have modified it. If the speculative thread is the first one to have modified it, the thread can go ahead and restore the value for that memory location. If the speculative thread is not the first one to modify it and aliasing on that location ($hash(x)$) is possible, the speculative thread has to check whether its memory location $x$ is actually contained within the write-set of the less speculative threads denoted in the shadow data structure for $hash(x)$. If it is not found in these less speculative write-sets, that speculative thread will restore the value for memory location $x$.

Once the memory state has been rolled back, the participating speculative thread can reset the pertinent memory locations in the shadow data structure in parallel.

### 3.7. Speculative Thread Lifecycle

Figure 4 is a state diagram illustrating the lifecycle of a speculative thread. A thread can be in one of the following states.

(1) *FREE*. Thread is ready to get the next chunk of iterations and start work.
(2) *RUN*. Thread has started speculative execution.
(3) *DONE*. Thread has finished execution. Note that since updates are in-place, the thread has actually committed its results (parallel commit). The thread now must wait until it becomes the least-speculative thread. While waiting it can still be squashed by a less speculative thread.
(4) *COMMITTING*. Thread is now the least-speculative thread and starts clearing its local data structures. At this stage the thread cannot be squashed since it has already finished execution and there are no less speculative threads.
(5) *COMMITTED*. Thread has cleared its local data structures and indicates that is ready to become a speculative thread again.
(6) *FAILED*. Thread has been involved in a data dependence violation.
(7) *SQUASHING*. Thread starts rollback in parallel with any other offending threads.
(8) *SQUASHED*. Thread has finished rollback. It is now waiting to be restarted.

## 4. LECTOR: SYSTEM DESCRIPTION

The following sections describe a software TLS runtime system written purely using the Java programming language.

### 4.1. General Idea

The runtime software system we propose in this article follows pessimistic concurrency control, lazy version management, and eager conflict detection. Parallel execution proceeds monitored by our TLS system. Performing a load or store, requires a thread to first acquire exclusive ownership of the desired memory location. Speculative stores are buffered locally only after ensuring that none of the more speculative threads has

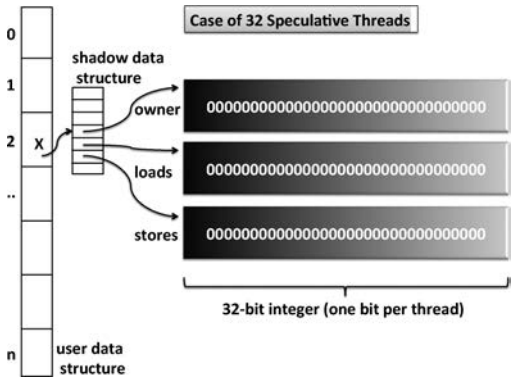


Fig. 7. Shadow data structure in Lector.

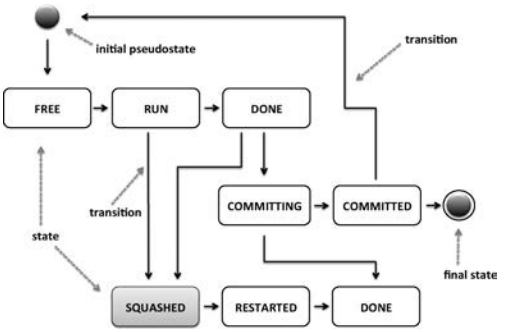


Fig. 8. Speculative thread lifecycle for Lector.

already loaded that location. Otherwise the more speculative threads are squashed due to RAW dependence violation. Speculative loads search the local buffer (write-set) first, before loading from memory, in case that value has already been written locally form an earlier store. This provides the illusion of processor consistency. If not found there, then the value has to be loaded from main memory. In case an earlier thread has written on that location, the current thread can either forward the most recent value, wait for the earlier thread to write-back its write-set to memory and then load, or squash. Buffered values are written-back to main memory after a thread has been proven successful.

### 4.2. Speculative Data Structures

The system utilizes a shadow data structure as exhibited in Figure 7. We present the case of 32 speculative threads in the system. Every user memory location is mapped into this shadow table using a hash function. Each mapping in the shadow table is composed out of three consecutive 32-bit integer memory locations as shown in the picture: One to represent the owner thread, one to indicate whether a thread performed a load, and one to indicate a store by a thread. A bit in any of this three locations reflects the identity of the thread that is accessing the original user's location for the appropriate action. The next section will provide more details on how that works in practice. Clearly, this example is for up to 32 threads but it can be easily expanded to greater number by allocating more memory for each mapping.

In addition to the shadow table, there is a local read- and write-set for each thread. The read-set is a list with all locations read by that particular thread, whereas the write-set is a hash map with all the location/value pairs to be written to main memory when that thread commits.

### 4.3. Speculative Operations

*4.3.1. Speculative Stores.* Whenever a speculative store takes place (see Figure 9), the thread must successfully acquire exclusive ownership of the location to be written in order to proceed. This is done by accessing the shadow table for that particular location (using a CAS operation) and setting the appropriate bit in the "owner" to indicate that this thread is operating on this location. If the thread finds the location locked, it spins a bounded number of times before initiating a squash. When the thread acquires exclusive ownership, it checks "loads" to see whether a more speculative thread has already loaded from that location. Such a load is called an *exposed load* in this context and requires all the more speculative threads to be squashed. There is no need to take any action if a less speculative thread has loaded that value or if there was a store

```
01. specStore(int hash, int threadID, Value v){
02.   if(shouldSquash()){thread.squash();}
03.   shadowStructure.lock(hash, threadID);
      //may throw exception and squash() here
04.   shadowStructure.checkForViolations();
05.   if( isNonSpeculativeThread(threadId) ){
06.      writeBackToMemory(v);
07.   }else{
08.      shadowStructure.markStore();
09.      writeSet.put(hash, v);
10.   }
11.   shadowStructure.unlock(hash, threadID);
12. }
```

Fig. 9. Speculative store.

```
01. specLoad(int hash, int threadID){
02.   if(shouldSquash()){thread.squash();}
03.   if( writeSet.contains(hash) ){
04.      return writeSet.get(hash);
05.   }else{
06.      shadowStructure.lock(hash, threadID);
         //may throw exception and squash() here
07.      shadowStructure.checkForViolations();
08.      shadowStructure.markLoad(hash, threadID);
09.      Value v = memory.loadValue();
10.      readSet.add(hash);
11.      shadowStructure.unlock(hash, threadID);
12.      return v;
13.
14.   }
15. }
```

Fig. 10. Speculative load.

by a different less speculative thread since this is a lazy version management system. Following the check for violations, the thread releases the lock and inserts the value in its write-set.

*4.3.2. Speculative Loads.* Since stores are buffered, a speculative load (Figure 10) will first check the thread-local write-set in case the value has already been written earlier by the same speculative thread. If this is true, the load will return the latest buffered value. There is no need for such a load to consult the shadow table since, it is guaranteed that the loaded value was produced by the correct store. If the value is not present in the write-set, then the thread acquires exclusive ownership of the location in the same manner as in speculative stores, checks for violations, and loads the value from main memory. There is no need to worry about different threads loading the same value as no conflict can arise from that. Also, threads from the future (more speculative) that have written to that location are harmless for the moment. The reason is because a more speculative thread that has produced a store in that location will buffer the value and write it back to memory when the time is appropriate. Then is when the system will worry about conflicts because it could be the case that the earlier thread was squashed. The only situation that can cause a problem is when a less speculative thread has produced a store for that location but not yet committed. That means that the current thread requires that value in order to proceed but that value is in earlier thread's buffer and not yet in memory. One solution (that is feasible due to our novel representation) is to identify, using the "stores" in the shadow table, the latest thread that has produced a store in that location (but not yet committed) and *forward* the correct value. This requires careful handling in case the write-set of the thread we forward from, is updated simultaneously. Another solution is to wait for the less speculative to write-back its results and load the value from memory. A simpler solution, which is the one we implement in our system, is to squash the current thread and restart its execution. If no violation is present the ownership is released, the value is loaded from main memory as normally, and the hash value from the shadow table is recorded in the read-set. The read-set is used when the thread commits in order to release the marking in the shadow table.

*4.3.3. Scheduling and Commits.* Our system follows a type of *scheduling window* (described in Section 2.2.4) for scheduling the threads. The window size is always the same as the number of threads; therefore, for $n$ threads we use a window of $n - 1$ size. Thread in slot 0 is the nonspeculative thread and similarly thread $n - 1$ is the most

speculative thread. Although thread 0 is allowed to write its results immediately back to memory (i.e., no buffering), it still checks the shadow table in order to eagerly squash more speculative threads that have exposed loaded from a location the nonspeculative thread updates. Threads in the window commit their results back to memory in ascending order of speculation by locking the location in the shadow table, propagating the appropriate values, and clearing the shadow table from their markings. After all threads in the window have committed, speculation restarts with all threads getting work from a work-queue.

### 4.4. Speculative Thread Lifecycle

Figure 8 is a state diagram illustrating the lifecycle of a speculative thread. A thread can be in one of the following states.

(1) *FREE*. Thread is ready to get the next chunk of iterations and start work.
(2) *RUN*. Thread has started speculative execution. Note that it can be switched to squashed, if it was involved in a data dependence violation.
(3) *DONE*. Thread has finished execution. Thread now must wait until it becomes the non-speculative thread in order to commit its results to memory. While waiting, it can still be squashed by a less speculative thread.
(4) *COMMITTING*. Thread is now the nonspeculative thread and starts clearing its local data structures and propagating the buffered values to memory. At this stage, the thread cannot be squashed since it has already finished execution and there are no less speculative threads. However, it can initiate a squash for a more speculative thread that has performed an exposed load form a location this thread is writing.
(5) *COMMITTED*. Thread has cleared its local data structures and finished any memory updates. It indicates that is ready to become a speculative thread again.
(6) *SQUASHED*. Thread has been involved in a data dependence violation. It must clear any marking in the shadow table.
(7) *RESTARTED*. Thread has re-started speculative execution.

### 4.5. Inspector Threads

We experiment with a novel technique in this work which combines the advantages of two popular models involved in the beginning of the TLS research, namely, *Inspector-Executor* model and `Lpd` *(Lazy Privatizing* DOALL*) Test* described in Rauchwerger [1998].

*4.5.1. Inspector-Executor Model.* Using the *Inspector-Executor* model, a simpler version of the loop under question is extracted and executed to verify whether the loop carries any data dependencies. The simpler version does not produce any side-effects as well as does not require all the code from the original loop (just the memory accesses). Therefore, the inspector is expected to execute faster than the original sequential loop. If proven safe, then the executor may execute the loop in parallel. Inspectors are created by analyzing memory accesses, collecting information about their iteration number and access type (read/write) in a separate data structure, and then checked for data dependencies [Saltz et al. 1991]. The drawback of this model is that, in cases where the loop cannot be stripped-down sufficiently, the inspector might end up taking the same time as the original loop.

*4.5.2.* `Lpd` *Test.* The `Lpd` test was more successful than the inspector-executor, being the heart of `Lrpd` [Rauchwerger and Padua 1995] test and `R-Lrpd` [Dang et al. 2002] test. The `Lpd` test checks the loop under question for data dependencies or whether dependencies can be eliminated when privatization is used (i.e., buffered updates). The test flags whether dependencies exist or not, while the loop executes in parallel buffer-

ing any updates to memory. If dependencies are found, the loop discards any buffered updates and executes sequentially; otherwise, the loop has been parallelized correctly using privatization. Lpd has an advantage over the *Inspector-Executor* model, especially when the inspector cannot strip-down the loop to an adequate level. A successful parallelization of Lpd in that case simply needs to update main memory, whereas the other model has to still run the executor. Still though, under both models, in case the loop was not found parallel, all execution and time spent while the test was running, will be wasted.

*4.5.3.* LPD *Meets Inspector-Executor.* In this work, we combine the two ideas in a way that their drawbacks are eliminated. We do extract a stripped-down version of the loop to be parallelized like in the case of inspector-executor; however, there is no executor as such. The inspector performs the Lpd test but since it does not replicate the entire code, it is expected to run faster. The inspector threads start running ahead, as soon as the application begins. At the same time, the loop is executed using our speculative parallelization runtime environment, described earlier in this section. Once the inspector phase completes, its results dictate how the speculation will continue. Given that the inspector found the loop to be DOALL, speculation is dropped, any buffered results are propagated to memory, and the loop continues to execute in parallel without any speculative overhead. The inspector terminates as soon as it finds a data dependency, allowing the execution to proceed speculatively. Moreover, even in the unfortunate case that the inspector finishes at the same time as the speculative execution, there is no need to execute the loop sequentially since it has been already executed speculatively. The same applies in case the loop contained any data dependencies. Using this scheme, DOALL loops that can express a "light" inspector, may benefit significantly from the absence of speculation-associated overhead.

Our system takes advantage of the benefit from the Inspector-Executor model that is discovering whether a loop is DOALL or not, quickly. At the same time, we eliminate its drawback of having wasted work in case the loop was not DOALL since our underlying TLS model executes alongside with the inspector. The same applies in case the inspector was as "heavy" as the loop itself. For the same reasons, we eliminate any shortcomings related with the Lpd test.

The Inspector threads in Lector are created following the PD test [Rauchwerger and Padua 1994] where a good example is provided of how to generate the inspector threads.

## 5. EVALUATION AND RESULTS

### 5.1. Evaluation Methodology

For all our experiments, we have used a UltraSPARC T2 system also known as Niagara 2. It has 8 processors, each of which has 8 hardware threads, making it able to process up to 64 threads simultaneously. Furthermore, it has 4-MB shared L2 cache. The Solaris®10 OS was installed on the machine. Solaris uses a "maximum dispersal" thread scheduling to assign threads to their initial processors. The kernel selects the least loaded core when placing a thread, in order to avoid resource contention among concurrently running threads. This is the default OS thread affinity and running a Java application it is not possible to change it using the standard libraries or JVM settings.

We have used Java™SDK version 1.6.0 and Java™HotSpot VM with fixed 4GB maximum heap size for all executions and the System.nanoTime() timer provided by the JVM. All the results presented are the average of ten executions for each thread number for each benchmark and we did check the standard deviation for statistical significance. The applications we have tested come mainly from two benchmark suits: SPECjvm2008 and JOlden. Following the methodology used by SpLIP [Oancea et al. 2009], we have also chosen applications that parallelizing their *loop-kernels* improves the total application

run time. SpLIP has been reimplemented in Java to provide a direct comparison with MiniTLS. We have also chosen applications with irregular accesses (not parallelizable by static compilers) thus making them good candidates for TLS. From SPECjvm2008 we experimented with: (i) Sparse, a matrix multiplication algorithm (the input used was the large data set), (ii) SOR, that simulates Jacobi Successive Over-Relaxation (again the large data set was used), and (iii) Monte-Carlo, which approximates the value of Pi (large data set used). From JOlden we experiment with: (i) Barnes Hut, an implementation of the Barnes-Hut force calculation algorithm (using input *C*), (ii) Em3d, a simulation of electro-magnetic waves traveling through objects in three dimensions, and (iii) Perimeter, an implementation of Samet's algorithm for computing perimeters of regions in a binary image. Finally we experiment with LeeRouter [Watson et al. 2007], a circuit routing using Lee's algorithm. For LeeRouter, the *mainboard* input dataset was used. Most of these applications are also used in previous TLS studies [Quiñones et al. 2005; Oancea et al. 2009; Tian et al. 2010; Ioannou et al. 2010].

As the focus of this article is on optimizing the software runtime system of TLS and not on how to transform a loop into its parallel equivalent by a compiler, the applications were transformed manually into parallel speculative versions. Two benchmarks exhibit dependencies (LeeRouter and Em3d), and five have not runtime dependencies. Loop-induction variables were eliminated as they introduce false dependencies under TLS. The most time consuming loops were considered for TLS parallelization. Finding these automatically is an interesting optimization problem in itself [Quiñones et al. 2005; Liu et al. 2006] (for task selection) and Johnson et al. [2004] and Ottoni et al. [2005] (for finding suitable tasks).

### 5.2. MiniTLS Experiments

*5.2.1. Baseline TLS system:* SpLIP. We compare MiniTLS against a state-of-art TLS library, SpLIP [Oancea et al. 2009]. The reason we have chosen SpLIP is because it also relies on eager memory version management and provides the best performance results. Thus, SpLIP sets an optimized baseline against to compare MiniTLS.

Although, MiniTLS and SpLIP use an eager version management technique, their implementations are fundamentally different. SpLIP employs two data structures for handling the iterations that load and store for each speculative access. Two additional data structures are used for imposing order to accesses to each location between reading and writing threads. Yet another data structure is required for storing a timestamp for a particular location in case of a rollback. The rollback procedure also differs from ours. SpLIP requires to aggregate the write-sets of all speculative threads involved in the violation by comparing timestamps in case of multiple thread access the same location. This requires some hash map in order to be able to check whether a location was already written by a different thread. In contrast, MiniTLS requires no aggregation, and each thread involved in the violation proceeds in parallel with each other for rollback, without using timestamp comparisons. Since all the information is kept in the main shadow structure and there is no need to check the write-set entries once they have been recorded, the write-set can be implemented as an array structure.

*5.2.2. Performance Results.* Figure 11 presents speedup results against the original sequential unmodified version, obtained from applying MiniTLS to the benchmark applications. The $y$ axis indicates speedup, whereas the $x$ axis shows the benchmarks used. The sequential application's speedup is marked by speedup $= 1$ in the $y$ axis, thus whatever is higher than that, shows improvement. On average, we start observing speedups when we go over 4 threads. The reason is due to the speculative overhead added by TLS. The parallelism starts amortizing the cost on average when more than 4, sometimes 8 threads are used. The cost introduced by speculative operations is high

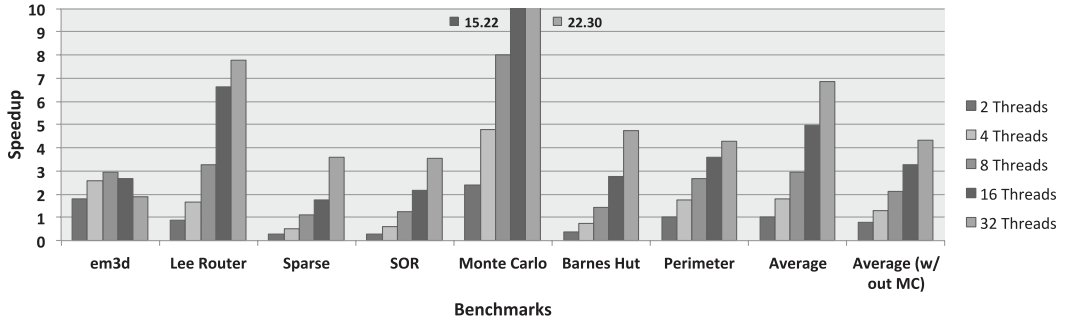


Fig. 11. Speedup results for MiniTLS. Sequential execution is denoted by 1 in the y axis.

and thus for small number of threads it drowns the benefits of parallelism. Figure 16 shows an example of the overhead introduced to support MiniTLS in terms of execution time for the Sparse application. The $y$ axis is intersected where the sequential time is (i.e., the baseline). When the overhead bars grow below the $x$ axis this implies overhead less than the sequential time, and thus the application starts showing speedups (after 8 threads). With 4 threads we have 1.8x faster than the sequential application, for 8 threads nearly 3x faster, 16 threads 5x faster and 32 threads almost 7x faster. In order to be objective, we have also included the average speedup excluding the Monte-Carlo benchmark, which does not carry any data dependences. Em3d shows a decline in speedup after 8 threads. The benchmark carries a large number of data dependences which causes more frequent rollbacks when more that 8 threads are used.

Figure 12 shows speedup comparisons between MiniTLS and SpLIP. As before, the $y$ axis shows speedup against the sequential unmodified application and the $x$ axis indicates the number of threads used. The same pattern is again observed in which speedups are observed after 4 or 8 threads (for the reasons explained earlier). MiniTLS outperforms SpLIP 1.33x, on average: 1.1x for 2 threads, 1.2x for 4 threads, 1.3x for 8 threads, 1.4x for 16 threads and 32 threads. The main reasons for this performance difference are analyzed in the next section where we present the execution overhead.

*5.2.3. Speculative Overhead Comparison.* Figure 12 shows how MiniTLS outperforms SpLIP in terms of speedup. The main reason why speculative systems suffer speedup losses is due to speculative overheads, which includes, for example, marking speculative loads/stores and time spent during rollback. MiniTLS shows performance improvements over its competitor by reducing those overheads. Figure 13 presents the reduction percentage of our system against SpLIP for speculative operations. The graph has two parts: The first part shows how much MiniTLS reduces speculative marking over SpLIP (SpLIP is the normalized baseline in all cases). The second part shows how much MiniTLS reduces rollback time over SpLIP (SpLIP is the normalized baseline in all cases). Both parts are independent. That is, Rollback percentage has nothing to do with the Marking percentage. For example, looking at the marking section for Em3d for 32 threads one can say that MiniTLS spends around 30% less time for marking compared to SpLIP (where 100% marking is the total amount of time for SpLIP to perform marking). We show LeeRouter and Em3d since they are the ones that carry data dependences and thus could see how much time is saved from rollback. Among the noncarrying dependency benchmarks we have chosen to show the average as a representative of all benchmarks (except Monte-Carlo) since they all feature similar execution patterns. The graph clearly shows the effectiveness of our rollback routine. This is due to two reasons: (i) SpLIP uses extra buffers for conservative synchronization before and after the same location is accessed by multiple threads. Due to the absence of locks the possibility of

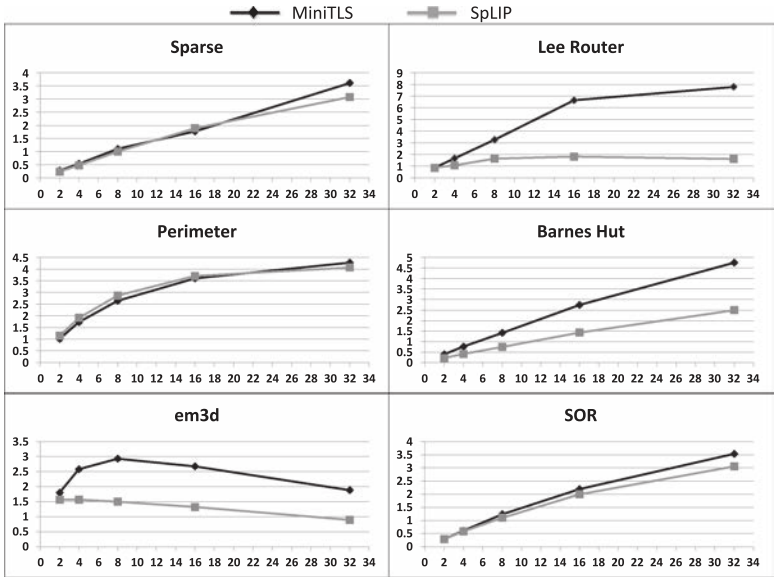


Fig. 12. Speedup comparison of MiniTLS and SpLIP.

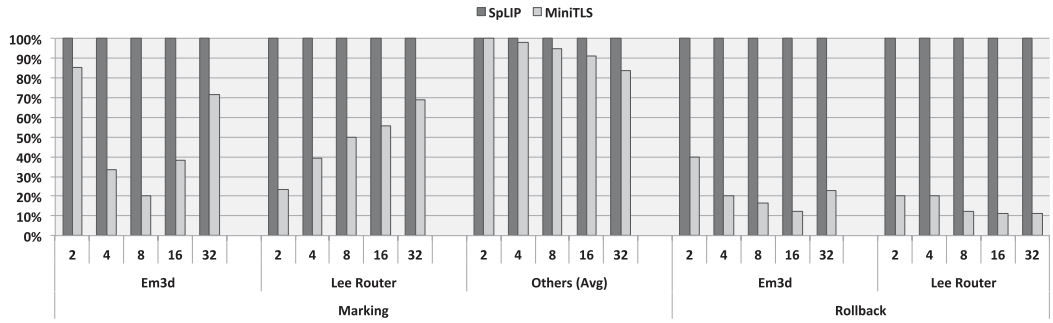


Fig. 13. Shows the amount of overhead reduction of MiniTLS against SpLIP. Graph is normalized (baseline SpLIP). First part shows reduction of Speculative read/write marking. Second part reduction of Rollback time.

those buffers to cause a data-race is very high when a particular location is accessed by multiple threads concurrently. (ii) When a violation occurs using SpLIP, the write-sets of the involved threads must aggregate their values, comparing their timestamps, before write-back occurs. In our implementation, this process is accelerated since the involved threads can proceed in parallel for write-back and without the cost of the timestamps. Marking for noncarrying dependency benchmarks is very similar among both systems. The absence of rollbacks allows both systems to spend approximately the same amount of effort to bookkeep their information. However, for the dependency-carrying benchmarks the case is different. Rollbacks cause a lot of code re-execution and thus more marking involved. Since MiniTLS performs less rollback, it benefits from having less code to re-mark. Barnes-Hut is the only application that does not follow the same pattern with other similar benchmarks especially after 8 threads. This application requires very minimal marking. When MiniTLS performs marking, it requires very little space compared to the other library as we explain in the following section.

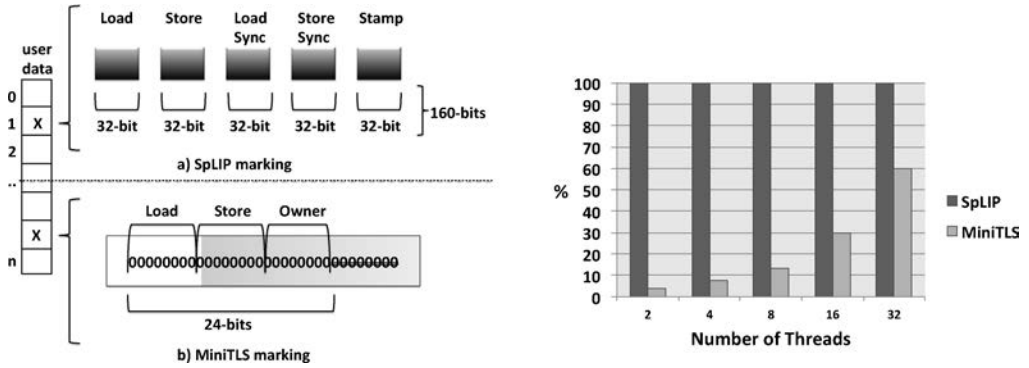


Fig. 14. Space required for 8 speculative threads using (a) SpLIP and (b) MiniTLS. (c) Normalized (baseline SpLIP) space overhead comparison between MiniTLS and SpLIP.

*5.2.4. Data Structures Space Comparison.* To maintain the speculative state, additional space is normally required. This section compares the space overhead between the two systems: MiniTLS and SpLIP. The space we refer in this section, is the space in regard to any data structures required for version management. We do not consider the space required by the undo log as there is negligible difference between the two approaches. Our novel marking scheme is designed in such a way as to require the minimal space based on the number of threads running (hence the name MiniTLS). A typical TLS system, unlike MiniTLS, will consume the same amount of speculative space regardless of the number of threads used. Figure 14(a) and Figure 14(b) illustrate the memory space required to facilitate speculative marking for eight threads for SpLIP and MiniTLS respectively. Figure 14(a) shows that for each user accessed memory location, SpLIP would require at most 160 bits in order to mark the iterations that will possibly perform a load or a store, the timestamp, as well as the thread ids for synchronizing loads and stores. Figure 14(b) shows that MiniTLS requires only 24 bits to perform the same operations as opposed to 160 bits that SpLIP requires. Figure 14(c) shows the normalized (with SpLIP as baseline) Space Overhead comparison between MiniTLS and SpLIP. There is a significant space overhead reduction of 96% when 2 threads are in use, 92% reduction with 4 threads, 87% with 8 threads, 70% with 16 threads, and 40% with 32. In other words, MiniTLS requires on average 5x less space than SpLIP. This can have a great impact in performance, especially in automatic memory managed languages such as Java since there will be less garbage collection triggers than normally required and thus less interruptions of the user's application.

*5.2.5. Memory Overhead.* While TLS can decrease application execution time, more memory is required to support the additional data structures. Apart from the single shadow structure, each thread has its own copy of its read and write sets. Thus, we conducted an experiment (similar to Tian et al. [2010]) to measure the memory consumption of TLS. Two representative applications were selected for comparisons. The first one, Sparse could be considered as the worst-case scenario as 90% of the total application accesses, are speculative. The second, Lee Router, is the average case application where about 50% of the total access are speculative. Figure 15 shows the results of comparing the additional memory required compared to the sequential application. For the "worst case", the memory consumed is between 0.2x and 1.8x for MiniTLS and 2.9x for SpLIP, compared to the sequential version. For the "average case" it was between 0.02x and 0.4x for MiniTLS and 0.6x for SpLIP, compared to the sequential version. What we considered as memory overhead in the experiment, is the total amount

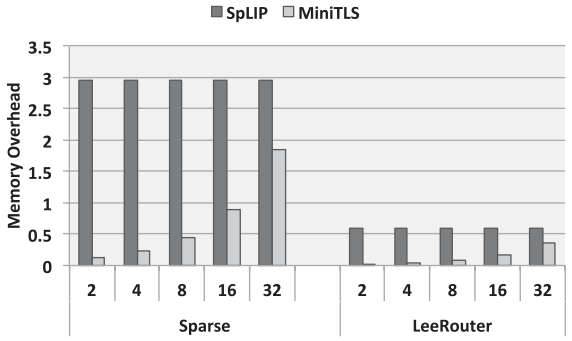


Fig. 15. Memory overhead of MiniTLS and SpLIP compared to the sequential version.

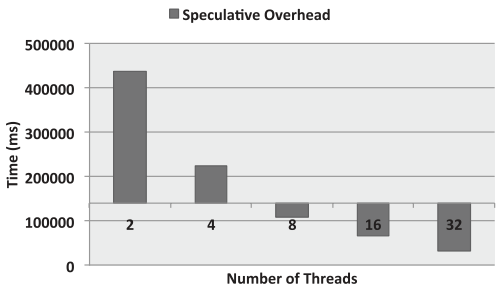


Fig. 16. Time spent on speculation for Sparse. The $y$ axis in intersected at the sequential time.

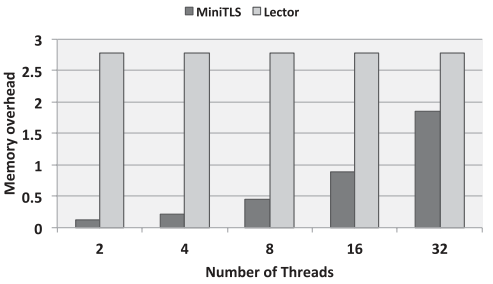


Fig. 17. Memory overhead of MiniTLS and Lector compared to the sequential version of the Sparse benchmark.

of storage in order to support the shadow data structure compared to the sequential application. For baseline memory overhead we took into account all the memory accesses that the sequential application performs during the entire execution. We measure the storage required for the shadow structure based on the unique speculative accesses during the execution of the application (only the unique since only one instance of a read/write exists in the shadow structure). Then we take into account how many bits are required for that information. SpLIP is constant across threads since the amount stored per memory location is always the same. For MiniTLS the storage requirements grow with the number of threads as more bits are necessary to support marking on those threads. Although not shown in the graph, there is also the extra overhead for maintaining local read/write sets. Storing these local per thread data structures requires 50% extra storage of the sequential application's memory requirements in these two benchmarks. This number is dependent on the proportion of data accessed during TLS execution, which in these benchmarks is fairly high. This is a constant overhead in TLS systems; however, it can be significantly optimized by replacing read-set data structures with *bloom filters*.

### 5.3. Lector - Experiments

For the following experiments, we compare three systems: TL2TLS, our baseline, which is explained next; LazyTLS, which is simply Lector with inspector threads disabled; and Lector, which has inspector threads enabled. When showing Lector for $n$ number of threads that means we execute $n$ inspector threads and $n$ TLS threads. Thus, for 2 threads, Lector uses 2 inspector threads and 2 TLS threads which are 4 threads in total.

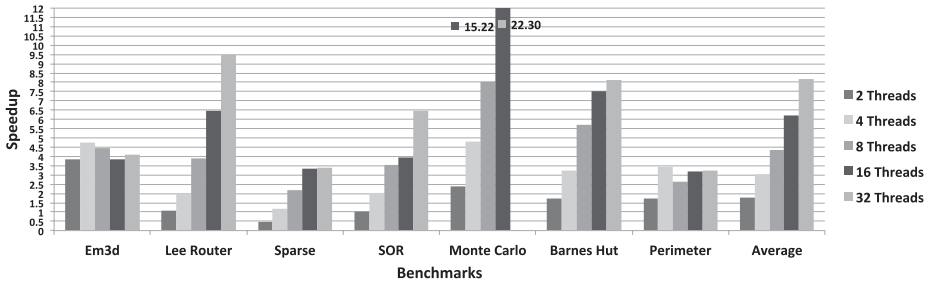


Fig. 18. Speedup results for `Lector` against the sequential execution. Sequential execution is denoted by 1 in the y axis.

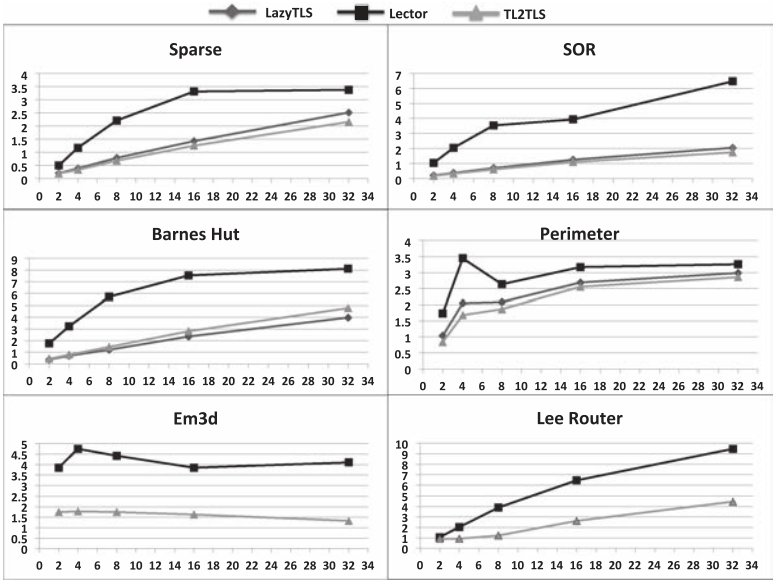


Fig. 19. Speedup comparison between `LazyTLS`, `Lector`, and `TL2TLS`.

*5.3.1. Baseline TLS system:* TL2TLS. In order to test the performance of our system we compare against an established baseline used in Mehrara et al. [2009] and Tian et al. [2010]. The baseline is based on the state-of-art algorithm - Sun's Transactional Locking 2 [Dice et al. 2006]. Similar to Mehrara et al. [2009] and Tian et al. [2010], speculative code is added within transactions and explicit synchronization is added into transactional functions to enforce in-order commit, which is necessary to maintain sequential program semantics. We will refer to the baseline as `TL2TLS` in the rest of the article.

*5.3.2. Performance Results.* Figure 18 presents speedup results against the original sequential unmodified version, obtained from applying `Lector` to the benchmark applications. On average, `Lector` performs between 1.8x and 8.2x faster among multiple threads compared to the sequential application. The following paragraphs explain how `Lector`'s inspector threads help minimize the speculative overhead and allow for more useful computation to be performed.

Figure 19 shows speedup comparisons between `LazyTLS`, `Lector`, and `TL2TLS`. As before, $y$ axis shows speedup against sequential unmodified application and $x$ axis indicates the number of threads used. For `em3d` and `Lee Router`, `LazyTLS`, and `Lector`

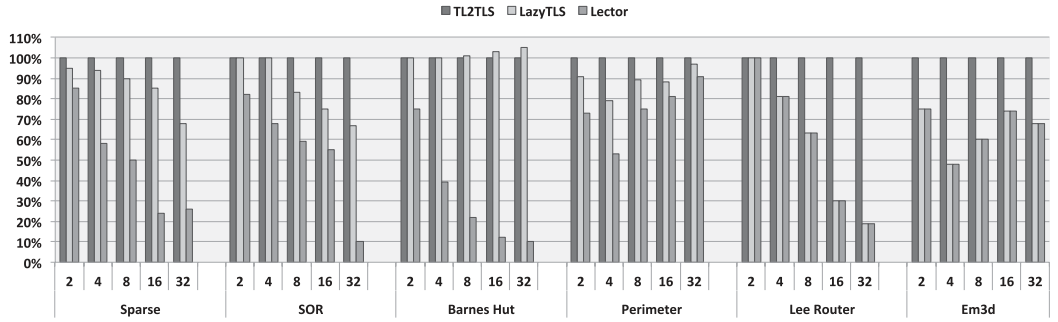


Fig. 20. Normalized speculative overhead reduction with baseline the TL2TLS system.

are actually identical (thus, only one is shown) because the inspection phase ends shortly after the first data dependency is found which for both benchmarks is nearly at the beginning of execution. We do not present speedup comparisons for Monte-Carlo since it does not carry any speculative activity and thus it has the same behavior under any system. The most interesting fact is the speedup benefits we observe in certain benchmarks when using the inspector threads. When inspection is enabled, even the minimal thread configuration produces higher speedup than running 32 threads with inspection disabled. For instance, in Sparse, SOR, and Barnes Hut it is more beneficial to run the benchmark with 2 speculative plus 2 inspector threads rather just with 32 speculative threads and no inspector threads. More details are explained in the following section where speculative overhead is discussed.

*5.3.3. Speculative Overhead Comparison.* Speculative overhead is maintaining the information in the shadow data structure as well as per thread TLS context support. Lector outperforms the other two configurations as it reduces that extra cost. Figure 20 shows the percentage of speculative overhead reduction between the three systems. The graph is normalized to TL2TLS. For a given amount of overhead that TL2TLS spends, we show the reduction for the other two systems. For example, executing Barnes-Hut with 8 threads, LazyTLS spends about the same time in speculative overhead as TL2TLS. Lector, on the other hand, spends about 80% less time compared to TL2TLS. In most cases TL2TLS and LazyTLS are similar, however in the cases of LeeRouter and Em3d (which are the ones that carry data dependences) our system is not only faster but also spends less time for speculative marking. This is mainly because TL2TLS performs lazy conflict detection (at commit time) which in case of many conflicts, produces wasted work and additional speculative marking. The idea of TL2 is that a successful thread will change the version of a memory location at commit time notifying any thread that holds an out-of-date version of the same location, to be squashed. However, a thread with an out-of-date version will not discover that incident before its commit time. Therefore, more execution and marking is done between the time a conflict occurs and the time of detection. In our implementation it is impossible for a thread to have an out-of-date version of a memory location and still keep executing. If a thread updates a given location, it will "see" in the shadow structure that a different thread has performed an action there and thus squash that thread eagerly.

In all cases, Lector has the lowest cost compared to the other two systems. In the noncarrying dependency benchmarks, an adequate inspector version was extracted and managed to finish earlier than speculative execution. Abandoning speculation (i.e., speculative marking) with the aid of inspector threads, allows speedup increase since there is no more speculative work after that point.

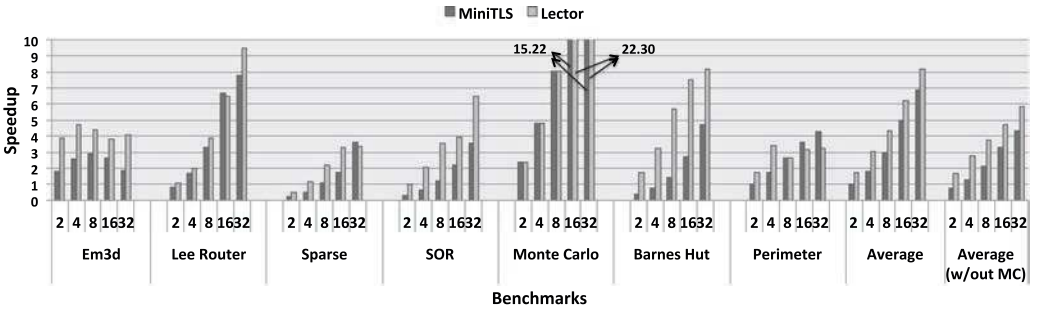


Fig. 21. Speedup results for `Lector` Vs. `MiniTLS` against the sequential execution. Sequential execution is denoted by 1 in the y axis.

`Lector` shows greater speedups as the number of threads increases. This is due to the advantages of the inspector threads during the marking phase having to check only their local speculative locations as opposed to the other two systems that using the same data structure for all thread configurations. `Perimeter` does not follow the same behavior due to the high number of speculative accesses in each inspector thread, causing the analysis phase to spend considerable time. `LeeRouter` and `Em3d` does not show any advantage using inspector threads since they carry data dependencies.

### 5.4. MiniTLS vs. Lector

Finally, we present a speedup comparison between our two systems: `MiniTLS` and `Lector`. As previously, we show speedups over the unmodified sequential execution for each benchmark. Figure 21 clearly shows how `Lector` outperforms `MiniTLS` nearly in every case. Inspector threads are able to quickly identify a loop without data dependencies and notify `Lector` that speculation is no longer necessary. Thereafter, the loop can execute in a fully parallel nonspeculative mode. In cases where data dependencies exist, inspector threads will not be of any additional benefit, however, they quit inspection as soon as conflicts are found to allow speculation to continue without any unnecessary overhead. Nevertheless, `Lector` is still faster in those cases. For the benchmarks we experiment with, `Lector` still benefits from its lazy version management as it does not trigger as many data dependencies as an eager version management system (lazy can only trigger RAW dependencies). On average `Lector` performs approximately 2x faster than `MiniTLS` for 2 and 4 threads, 1.8x faster for 8 threads, and nearly 1.5x faster for 16 and 32 threads. `MiniTLS` appears to be faster than `Lector` on the `Perimeter` benchmark for 16 and 32 threads. The reason for this is the relatively high speculative overheads of `Lector` for `Perimeter` running with 16 and 32 threads, shown in Figures 19 and 20.

Inspector threads can only be applied to systems that buffer their memory updates. Such threads cannot be applied to eager version management systems such as `MiniTLS` as updates to main memory are performed in-place and the inspectors can load the wrong values and by extension addresses.

The results indicate that `Lector` outperforms `MiniTLS` in most cases. However, this comes with the cost introduced by the additional inspector threads in terms of memory consumption. Figure 17 shows the memory overhead of both systems for the `Sparse` benchmark against the sequential version. `Lector` requires 1.5x more memory than `MiniTLS`. That is, 2.7x compared to the sequential version. Nevertheless, this is acceptable as it has still lower overhead compared to `SpLIP` (as shown in Figure 15) and it is faster.

## 6. RELATED WORK

One of the earliest software TLS systems was the Lrpd test [Rauchwerger and Padua 1995] later extended to R-Lrpd [Dang et al. 2002]. In this system, the tests for conflicts are performed only at the end of speculative execution creating a surplus of wasted work (if unsuccessful) and delaying conflict resolution. They use a sliding window scheduling policy to minimize memory overheads, however, the window moves only when all threads that are currently in-flight finish. Commits are serial, which hinders scalability. In addition there is contention at the end of each speculative thread, in which threads are trying to merge their results with the global data structures to perform the test. Cintra and Llanos [2005] improve R-Lrpd using a more dynamic thread partitioning scheme as well as a window that slides upon every thread commit rather than only when all threads in the window have finished. Forwarding is supported, but when enabled, unnecessary checks are introduced on applications that perform many reads first and minimal writes. Commits are serialized as in R-Lrpd, and when checks for violation occur at the end of a speculative section it is necessary to check at least all the exposed-loads by all the threads involved. Again, this may slow down significantly applications with many readers and few writers. Conflicts occur by pointer aliasing are not addressed. In MiniTLS, we employ parallel commits, which allow higher scalability, and forwarding is implicit avoiding any backward checks on exposed loads. Like Cintra and Llanos, we also perform eager conflict detection. In Lector, when forwarding is enabled, the thread that requires the value can look at the shadow structure and know immediately whether a thread has produced that value and exactly which thread. Like Cintra and Llanos, we also perform eager conflict detection.

Perhaps the most similar system to MiniTLS is the in-place implementation proposed by Oancea et al. [2009]. In their solution, serial commits are avoided and *CAS* instructions or memory fences are not used. The main difference from our system is in the way that they perform the version management. Two data structures are used, one for loads and one for stores, as well as two additional data structures to facilitate data race prevention, since locks are absent. An extra data structure is required to store timestamps. Thus, for every memory location check, they need to obtain information from at least four different data structures. Our solution minimizes significantly the space overhead by using a more compact bitwise solution, storing all the required information in just one data structure. In addition, we utilize a parallel rollback mechanism as opposed to the sequential used in their work, which also boosts performance. Experimental performance comparison results among Oancea et al. [2009] and MiniTLS were presented in Section 5. Lector uses buffered updates, which can only cause *RAW* violations, in contrast to in-place updates, which will cause the system to rollback in case of WAR and WAW data dependencies in addition to *RAW*.

Mehrara et al. [2009] present STMLite, a STM model modified to support speculative parallelization. STMLite aims to reduce the overhead associated with validating the read-set by decoupling the conflict detection from the main process using a central commit manager. Also individual locks for copying out the write-set are avoided. The main problem with such a solution is that maintaining a centralized point of control will harm scalability when the number of threads increases. Furthermore, during validation, a global clock is incremented to maintain consistency. Since the global clock is incremented at commit-time, like in TL2 [Dice et al. 2006], the system might end up performing unnecessary validations. Tian et al. [2010] describe a system, based on their earlier work; CorD [Chen et al. 2008], for supporting speculative parallelization for dynamic data structures. Their system separates speculative (shared across threads) from nonspeculative state (local to each thread). When a speculative thread is created, any value that is needed, is copied-in from the nonspeculative state to the speculative

one and later the results are copied back if speculation was successful. A mapping table is maintained for each speculative thread that has entries associated with each variable copy. Version numbers are used between the two states in order to detect misspeculations. Work from speculative threads is committed in-order by the main thread. A number of optimizations are also described to reduce overheads for their copying mechanisms. While this work is attractive, especially for applications that manipulate dynamic data structures, the extra mapping tables, and the copying of values is likely to hinder performance in the applications we are interested in this work. We aim to improve over those solutions in terms of scalability due to our parallel commits (in the case of `MiniTLS`) as well as space overhead due to our compact data structures.

Finally, Raman et al. [2010] propose the use of software multithreaded transactions for speculative parallelization. In this case, each thread executes part of the loop for all iterations and threads are scheduled is such a way to form a pipeline. This is in contrast to conventional TLS systems where entire loop iterations are distributed across multiple threads. Their system uses lazy version management and lazy conflict detection. The focus of their work is on software pipelining parallelization rather than DOALL loops.

There is also numerous work in a very closely related research area, *Software Transactional Memory* (STM) [Dice et al. 2006; Dalessandro et al. 2010; Spear et al. 2009b; Felber et al. 2008; Dice and Shavit 2010]. In *Transactional Memory* (TM), parallel applications are executed concurrently as transactions, and access shared data simultaneously. Underlying mechanisms guarantee correct execution of those transactions. TM shares most of the semantics with TLS except from the fact that threads in TM need not to be ordered. Saha et al. [2006] show a comparison between lazy and eager version management. They demonstrate that eager updates perform better than lazy updates. They find most of the lazy system's overhead coming from the fact that every time a value need to be loaded, the thread local buffer needs to be searched whether the value is already there. On the other hand, Spear et al. [2009a] argue that an STM system is better-off using a lazy strategy. Other studies on STM, that experiment with both version management systems, include Wang et al. [2007] and Felber et al. [2008]. Rock, developed by Sun Microsystems®, was intended to be the first general-purpose processor to support TM, but it was never commercialized. However, TM finally is making its way to hardware as part of the Intel®Haswell processor and Blue Gene/Q [Wang et al. 2012]. Furthermore, Intel provides experimental compiler support in `gcc` and `C++` language constructs for TM [Ali-Reza et al. 2012]. These models will open new research paths for TLS, since they can be used to accelerate speculative operations.

The design of our compact software data structure closely resembles the techniques used for hardware directory-based cache coherence schemes [Culler et al. 1998]. Those systems rely on a directory to keep track of all processors caching a memory block. A basic implementation for small number of processors keeps a bit vector per memory block, comprised of one bit per processor. The downside of this approach is the large memory requirements to keep the information as the number of processors increases. Empirical studies showed that the blocks kept in each processor's cache is relatively small compared to all blocks in main memory. Thus, more efficient implementations keep a small number of pointers (to the processors that have the block) per directory entry [Agarwal et al. 1988]. Techniques were also proposed to handle situations when the size to keep the number of pointers overflows [Gupta et al. 1990]. Our compact data structure is similar in a way since both ideas need to record readers/writers in a bit vector. Our structure differs on the fact that the bits also represent an ordering (the speculation order) besides ownership. Furthermore, our implementation choice is not concerned with scaling issues, since we target systems with small number of processors.

## 7. CONCLUSIONS

A significant barrier for adopting software TLS is the overheads associated with maintaining speculative state.

Two techniques for version management have been used extensively in the literature. For completeness, we propose two systems, one for each technique and we show that in both cases we are better than the state-of-the-art.

We have proposed a software TLS system with a novel compact version management representation; `MiniTLS`. Facilitated by this representation, `MiniTLS` reduces the space overhead over state-of-the-art software TLS systems between 96% on 2 threads and 40% on 32 threads. `MiniTLS` relies on eager memory data management and, thus, when a misspeculation occurs a rollback process is required. `MiniTLS` takes advantage of the novel compact version management representation to parallelize the rollback process and is able to recover from misspeculation faster than existing software eager TLS systems.

We also propose a second TLS system, `Lector` (`Lazy inspECTOR`), which uses a novel way of minimizing speculative marking (also uses the compact version management data structure). `Lector` performs on average 1.7x faster for 32 threads over an established state-of-the-art software TLS system. While the conventional TLS system is running, lightweight inspector threads are executed alongside to verify quickly whether speculative state maintenance is actually required. Those threads are highly likely to be faster than the TLS threads, as they only inspect a stripped-down version of the actual loop iteration consisting of memory accesses. Should the inspector threads discover that the loop is DOALL, speculation is abandoned allowing the application to run in a parallel speculation-free mode. On the other hand, if the inspector exposes any dependencies, then inspection is terminated and the system continues with conventional speculative parallelization.

We have applied `MiniTLS` and `Lector` to seven Java sequential benchmarks (with presence of misspeculations for two benchmarks), including three benchmarks from `SPECjvm2008`. The experiments for `MiniTLS` report average speedups of 1.8x for 4 threads increasing close to 7x speedups with 32 threads. `Lector` experiments report average speedups of 1.7x for 2 threads increasing close to 8.2x speedups with 32 threads. We have shown that TLS can speedup the execution of some `SPECjvm2008` benchmarks but we have not fully explored which other `SPECjvm2008` will benefit from TLS. Similarly, for `Lector`, we have not explored “informed" scheduling to avoid discovered dependencies.

## REFERENCES


AGARWAL, A., SIMONI, R., HENNESSY, J., AND HOROWITZ, M. 1988. An evaluation of directory schemes for cache coherence. In *Proceedings of the International Symposium on Computer Architecture*. 280–298.

ALI-REZA, ADL-TABATABAI, SHPEISMAN, T., AND GOTTSCHLICH, J. 2012. Draft specification of transactional language constructs for C++. Tech. rep., Transactional Memory Specification Drafting Group.

CHEN, M. K. AND OLUKOTUN, K. 2003. The JRPM system for dynamically parallelizing Java programs. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*. 434–446.

CHEN, T., MIN, F., NAGARAJAN, V., AND GUPTA, R. 2008. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the International Symposium on Microarchitecture*. 330–341.

CINTRA, M. AND LLANOS, D. 2005. Design space exploration of a software speculative parallelization scheme. *IEEE Trans. Paral. Distrib. Syst. 16,* 6, 562–576.

CINTRA, M., MARTÍNEZ, J. F., AND TORRELLAS, J. 2000. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*. 13–24.

CULLER, D., SINGH, J., AND GUPTA, A. 1998. *Parallel Computer Architecture: A Hardware/Software Approach*.

DALESSANDRO, L., SPEAR, M. F., AND SCOTT, M. L. 2010. NOrec: streamlining STM by abolishing ownership records. In *Proceedings of Symposium on Principles and Practice of Parallel Programming*. 67–78.

Dang, F., Yu, H., and Rauchwerger, L. 2002. The R-LRPD Test: Speculative parallelization of partially parallel loops. In *Proceedings of International Parallel and Distributed Processing Symposium*. 20–29.

Dice, D., Shalev, O., and Shavit, N. 2006. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC 2006)*. 194–208.

Dice, D. and Shavit, N. 2010. TLRW: return of the read-write lock. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*. 284–293.

Felber, P., Fetzer, C., and Riegel, T. 2008. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*. 237–246.

Gupta, A., Weber, W.-D., and Mowry, T. C. 1990. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Proceedings of the International Conference on Parallel Processing (ICPP)*. 312–321.

Ioannou, N., Singer, J., Khan, S., Yiapanis, P., Pocock, A., Xekalakis, P., Brown, G., Luján, M., Watson, I., and Cintra, M. 2010. Toward a more accurate understanding of the limits of the TLS execution paradigm. In *Proceedings of the IEEE International Symposium on Workload Characterization*.

Johnson, T., Eigenmann, R., and Vijaykumar, T. 2007. Speculative thread decomposition through empirical optimization. In *Proceedings of the Principles and Practice of Parallel Programming*. 205–214.

Johnson, T. A., Eigenmann, R., and Vijaykumar, T. N. 2004. Min-cut program decomposition for thread-level speculation. In *Proceedings of the on Programming Language Design and Implementation*. 59–70.

Kim, H., Johnson, N., Lee, J., Mahlke, S., and August, D. I. 2012. Automatic speculative DOALL for clusters. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*.

Kim, H., Raman, A., Liu, F., Lee, J. W., and August, D. I. 2010. Scalable speculative parallelization on commodity clusters. In *Proceedings of the International Symposium on Microarchitecture (MICRO'43)*. 3–14.

Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., and Torrellas, J. 2006. POSH: A TLS compiler that exploits program structure. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles of Parallel Programming*. 158–167.

Luo, Y., Packirisamy, V., Hsu, W.-C., Zhai, A., Mungre, N., and Tarkas, A. 2009. Dynamic performance tuning for speculative threads. In *Proceedings of the International Symposium on Computer Architecture*. 462–473.

Madriles, C., Lopez, P., Codina, J. M., Gibert, E., Latorre, F., Martinez, A., Martinez, R., and Gonzalez, A. 2009. Anaphase: A fine-grain thread decomposition scheme for speculative multithreading. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 15–25.

Mehrara, M., Hao, J., Hsu, P.-C., and Mahlke, S. 2009. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 166–176.

Oancea, C., Mycroft, A., and Harris, T. 2009. A lightweight in-place implementation for software thread-level speculation. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*. 223–232.

Ottoni, G., Rangan, R., Stoler, A., and August, D. I. 2005. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the International Symposium on Microarchitecture*. 105–118.

Packirisamy, V., Zhai, A., Hsu, W.-C., Yew, P.-C., and Ngai, T.-F. 2009. Exploring speculative parallelism in SPEC2006. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. 77–88.

Prabhu, M. K. and Olukotun, K. 2005. Exposing speculative thread parallelism in SPEC2000. In *Proceedings of the International Symposium on Principles and Practice of Parallel programming*. 142–152.

Quiñones, C. G., Madriles, C., Sánchez, J., Marcuello, P., González, A., and Tullsen, D. M. 2005. Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*. 269–279.

Raman, A., Kim, H., Mason, T. R., Jablin, T. B., and August, D. I. 2010. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*. 65–76.

Rauchwerger, L. 1998. Run-time parallelization: Its time has come. *Paral. Comput. 24*, 527–556.

Rauchwerger, L. and Padua, D. 1994. The privatizing DOALL test: A run-time technique for DOALL loop identification and array privatization. In *Proceedings of the 8th International Conference on Supercomputing (ICS'94)*. 33–43.

Rauchwerger, L. and Padua, D. 1995. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI'95)*. 218–232.

Saha, B., Adl-Tabatabai, A.-R., Hudson, R. L., Minh, C. C., and Hertzberg, B. 2006. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*. 187–197.

SALTZ, J. H., BERRYMAN, H., AND WU, J. 1991. Multiprocessors and run-time compilation. *Concurr. Practice Exper.* 573–592.

SPEAR, M. F., DALESSANDRO, L., MARATHE, V. J., AND SCOTT, M. L. 2009a. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)*. 141–150.

SPEAR, M. F., SHRIRAMAN, A., DALESS, L., AND SCOTT, M. L. 2009b. Transactional mutex locks. In *Proceedings of the SIGPLAN Workshop on Transactional Computing*.

TIAN, C., FENG, M., AND GUPTA, R. 2010. Supporting speculative parallelization in the presence of dynamic data structures. In *Proceedings of the Symposium on Programming Language Design and Implementation*. 62–73.

WANG, A., GAUDET, M., WU, P., AMARAL, J., OHMACHT, M., BARTON, C., SILVERA, R., AND MICHAEL, M. 2012. Evaluation of blue Gene/Q hardware support for transactional memories. In *Proceedings of the Symposium on Parallel Architecture and Compilation Techniques*.

WANG, C., CHEN, W.-Y., WU, Y., SAHA, B., AND ADL-TABATABAI, A.-R. 2007. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'07)*. 34–48.

WATSON, I., KIRKHAM, C., AND LUJAN, M. 2007. A study of a transactional parallel routing algorithm. In *Proceedings of the Symposium on Parallel Architecture and Compilation Techniques*. 388–398.

Received June 2012; revised September 2012; accepted November 2012