specSTM: Software Transactional Memory with Thread-Level Speculation Support

Daniel Pinto
daniel.pinto@ist.utl.pt

Instituto Superior Técnico, Lisboa, Portugal

May 2015

Abstract

With multi-core machines becoming increasingly more common, new approaches are needed to take advantage of the increasing number of cores. Transactional Memory and Thread-Level Speculation have been studied as approaches to this problem. With this article we obtain promising results by combining both approaches.
Keywords: transactional memory, thread level speculation, parallel computing, multi-core systems

1. Introduction

With multi-core CPUs becoming mainstream, it becomes increasingly important for the programmers to write programs that take advantage of multiple cores. However, writing these programs is often a difficult task.

To simplify the creation of muti-threaded programs, several approaches exist. One of them is Transactional Memory (TM) [7]. To use TM, the programmer identifies the blocks of the program that must run atomically and encloses them in an atomic block. These blocks will be run atomically by the TM system. However, the programmer still needs to be sure that the atomic blocks are commutative for every possible execution order. Otherwise, the program will behave incorrectly. This can make the programmer follow the conservative approach of only forking new threads for large blocks of code. Because of this, most TM programs are still organized as a small number of coarse-grained threads. This is evidenced by some applications [13, 3].

Another paradigm is Thread Level Speculation (TLS) [12, 4] which, automatically parallelizes the program by running tasks speculatively in parallel. When a violation of the serial program order is detected, the changes made by the task are discarded and the task is run again. However, TLS can only achieve positive results when the number of conflicts is very low (usually less than 1%) [10]. Most of the programs have too many conflicts to be parallelized effectively [11].

Instead of choosing between TM or TLS, we can combine TM and TLS in a single system to increase parallelism. Recent research [1] has shown that this approach can achieve promising results. Despite the positive results, this approach has several limitations. While it provides a new algorithm to join TM and TLS, the design is not modular and, as such, cannot take advantage of new research advances without developing a whole new algorithm. However, developing a new algorithm requires a large amount of effort. We believe that there are several benefits in having a solution that allows the programmer to choose any TM and TLS systems and join them together almost effortlessly. This is what we propose with this article.

2. Background

In this section, we detail some important aspects of TM and TLS systems.

Thread-Level Speculation

The main goal of TLS is to provide improved performance for applications not originally designed to make use of multiple cores. This is achieved by automatically splitting a sequential program into multiple threads that will run in parallel. Threads are created speculatively to run blocks of code that are likely independent. When a a conflict happens, the system detects it and resolves it at run-time, ensuring that the end result is the same as if the code had run in the original sequential order. Because the threads running in a TLS system have as specific order (the same as the serial program), the thread that will abort is always the most speculative one. Although the existence of conflicts does not affect the correctness of the parallelized pro-

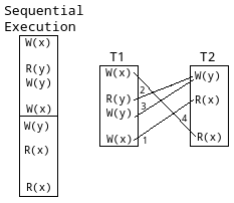

Figure 1: Types of possible dependences. R(x) means read from memory location x and W(x) means write to memory location y. (1) and (4) are flow dependences, (3) is an output dependence and (2) is an anti-dependence

gram, due to the cost of rollbacks, conflicts should still be avoided.

Dependences

There are two types of dependences: control dependences and data dependences.

Control dependences happen when the control flow of the code depends on previous code. When running code speculatively, it is possible to run code that would not have run in the sequential order resulting in a control dependence violation. An example of this happens when using loop-level speculation. A loop iteration may be started speculatively and a previous iteration may cause the loop to exit prematurely, before reaching the more speculative iteration. When this happens, the most speculative iteration should be rolled-back but should not be rerun, since it should not have run in the first place.

Another type of dependences are data dependences. Data dependences happen when different code writes to the same memory locations. When code that accesses the same memory location runs in parallel, a conflict may occur. There are three types of data dependences, illustrated in [Fig. 1]. When an instruction depends on the result of a previous instruction, it is called a flow dependence. When an instruction requires a value that is later updated, it is called an anti-dependence. When the order of the instructions will affect the result of a variable, it is called an output dependence. When one of these dependences are not respected, we have a read-after-write (RAW), write-after-read (WAR) or write-after-write (WAW) hazard respectively [6].

Spawning Threads

One of the approaches to spawn new threads is to use loop-level speculation. In loop-level speculation new threads are created when a loop is reached, each thread running one or more iterations of the loop. An example of how this division could be made is shown in [Fig. 2]. A system that uses this approach is SpLip [10].

```
for(i = 0; i < 10; i++) {
    foo(i);
}
```

```
Thread 1    Thread 2
 foo(0);     foo(5);
 foo(1);     foo(6);
 foo(2);     foo(7);
 foo(3);     foo(8);
 foo(4);     foo(9);
```

Figure 2: Example of code parallelized using loop-level speculation

Transactional Memory

In order to use transactional memory, the programmer starts by identifying the blocks of code that must run atomically and makes them run inside a transaction. A transaction is defined as a sequence of actions that appears indivisible and instantaneous to an outside observer [5]. In a TM system the outside observer would be the other transactions executing in parallel. A transaction can either end by committing and making all of its changes visible to other transactions or by aborting and ensuring that no effects of the transaction are visible to other transactions. When a transaction aborts, it has to be run again until it succeeds in order to achieve the exactly-once execution semantics that the programmer expects.

Design Choices

Although every TM system has to provide the guarantees enunciated above, there are several ways to do so. In this section we look into the choices that can be made when implementing a TM system.

Concurrency Control

TM systems must detect and resolve conflicts that occur between transactions. There are two main approaches to deal with the occurrence of conflicts.

The first one is pessimistic concurrency control. When using this approach, the conflicts are resolved immediately after being detected. The resolution of the conflict can be either aborting or delaying one of the transactions that caused the conflict. Care must be taken to avoid deadlocks. Deadlocks can be avoided by acquiring access in a fixed, predetermined order, or by using timeouts or other dynamic deadlock detection techniques to recover from them [8]. In environments where conflicts are frequent, this approach can be the best choice.

The other approach is optimistic concurrency control. When using this approach, the resolution of the conflict can happen after the conflict is detected. This allows the conflicted transactions to keep running, but the TM system must resolve the conflict before they are committed. Conflicts can be resolved by delaying or aborting one of the conflicting transactions. Care must be taken to avoid the situation where two transactions keep conflicting with each other, not allowing any of them to make progress. This is called a livelock. In envi-

ronments where conflicts are rare, using optimistic concurrency control can lead to better performance than using pessimistic concurrency control due to avoiding the costs of locking.

It is also common to mix the two approaches, using different approaches for different types of conflicts such as using pessimistic concurrency control on write-write conflicts and optimistic concurrency control on read-write conflicts.

Version management
Because a transaction can abort before completion and its effects must not be visible, TM systems must provide a way to undo any write that a transaction may have done. Once again, there are two different approaches used to approach this problem.

A TM system can use eager version management, in which case the transactions write directly to memory and maintain and undo-log with the values that were overwritten. In case of abort, the values from the undo-log are written back to memory. Eager version management is only compatible with pessimistic concurrency control. This is because the transaction needs exclusive access to the memory.

The opposite approach is using lazy version management, in which the transaction writes to a redo-log instead of writing directly to memory. When reading from memory, this log is checked to see if the location was written by the transaction before. If it was, the value is returned from the redo-log. If the transaction aborts the redo-log is simply discarded. If, otherwise, the transaction commits, the memory locations in the redo-log are updated with the new values.

Conflict detection
The way TM systems detect conflicts depends on the concurrency control they use. When using pessimistic concurrency control, conflict detection is already taken care of, because pessimistic concurrency control uses a lock which cannot be acquired if another transaction already has acquired it. This prevents conflicts by providing exclusive access to memory for the transaction that acquires the lock.

However, when using optimistic concurrency control, there is a wide variety of techniques that may be used. In these systems, there is usually a validation operation that will check if the current transaction has experienced a conflict or not. If no conflict happened, the validation will succeed, meaning that the transaction execution up until that point could have occurred in a legitimately serial execution.

There are three different dimensions to categorize conflict detection:

- Eager vs Lazy - In eager conflict detection, conflicts are detected when a transaction signals its intent to access data. If the conflict detection happens at commit, it is called lazy conflict detection. Conflicts may also be detected on validation, which does not necessarily happen only on commit. Validation may occur at any point of the transaction and it my happen more than one time.

- Tentative vs Committed - If a system detects conflicts between several running transactions, it uses tentative conflict detection but if it detects only conflicts between active transactions and transactions that have already committed, it uses committed conflict detection.

- Granularity - granularity is the size of data that is used to detect conflicts. Some hardware TM systems may detect conflicts at the level of cache lines while software TM systems may detect conflicts at higher levels, such as at the object level. Most TM systems will have some situations where a false conflict is detected: two transactions access different memory locations, and the TM system wrongly detects a conflict.

Generally, eager mechanisms are used with tentative conflict detection and lazy mechanisms are used with committed conflict detection. Similarly to concurrency control, two approaches may be used together, using one approach for read-write conflicts and another for write-write conflicts.

3. Implementation

Before getting into more detail, it is important to define some terms used throughout this chapter:

- Shared variable, location or memory - memory location that is accessed by transactions running concurrently. These variables need synchronization (provided by TM or by another way) in their accesses to ensure correctness.

- TLS parallelized code - code that is run by the threads created by the TLS system.

When considering the problem of joining TM and TLS like black boxes, a first naive approach would be simply to run TLS on top of the TM threads without any modification. This approach fails in several ways. The correctness of conventional TLS algorithms relies on the assumption that the underlying (single-threaded) program exclusively accesses thread-local variables. This assumption no longer holds once we add TM to our program. On the other hand, TM systems rely on the fact that all shared memory accesses go through TM functions. This assumption also fails once we add TLS. When we write to a shared memory location inside TLS parallelized code, the write and all further accesses to that variable must be made through the

TLS functions instead. Otherwise, correctness of the TLS parallelized code is not ensured.

An explanation of the problems we encountered along with our initial approach to resolve them are illustrated next.

The main problem resides in the fact that if a transaction aborts while inside the TLS parallelized code, the whole transaction will be restarted without running any further code. This means that the code that is responsible for rolling-back any changes made by TLS will not run. If the TLS system uses in-place writes this will leave the shared memory in an inconsistent state. Additionally, the internal data used by the TLS system will also be left in an inconsistent state. Although TLS systems have rollback functions they are used to abort the speculative tasks that are known to be wrong, when a conflict is detected. The rollback function does not abort the whole TLS code and is not meant to be called by the user. This means that we have no way of rolling back the changes made by TLS when an abort occurs. As such, our only option is to guarantee that no abort happens when inside TLS code.

Since we cannot prevent the TM functions from aborting a transaction, the only way to prevent aborts inside the TLS parallelized code is to avoid calling these functions. However, we still need to ensure the correctness of the read and written shared variables. Since we are running these transactions without the TM system (that would detect conflicts and rollback transactions where needed), we need to avoid conflicts between transactions that run without TM. This is done by limiting the possible combinations of concurrent transactions. To help us reason about the possible combinations of transactions that can run concurrently, we categorized the possible transactions according to two dimensions: having TLS sections and writing to shared locations. These gives us into four types of transactions:

- TLS-RW - Has TLS sections. Writes to shared locations.
- TLS-RO - Has TLS sections. Does not write to shared locations.
- SimpleRW - Does not have TLS sections. Writes to shared locations.
- SimpleRO - Does not have TLS sections. Does not write to shared locations.

In order to ensure correctness, the different types of transactions could be able to run concurrently according to table [Table 1].

This restrictions are ensured by the use of locks at the beginning and end of the transactions. This allows to run transactions with TLS parallelized code without having to modify either the TM or TLS algorithms.

		Simple		**TLS**	
		RO	**RW**	**RO**	**RW**
Simple	**RO**	X	X	Same as bottom left	
	RW	X	X		
TLS	**RO**	X		X	
	RW				

Table 1: Types of transactions supported concurrently on the initial approach. X means that the transactions are allowed to run concurrently. The upper right part would be the same as the bottom left since they are both comparing Simple transactions with TLS transactions (the order does not matter).

However, as it can be seen from the table, the only case where a TLS-RW transaction would be able to run is when nothing else is running. Additionally, the TLS-RO transactions would also be very restricted, since they would only run when no other transactions performing writes would be running. These restrictions would seriously hinder parallelism. This lead us to conclude that, the performance of this model would be very poor.

Proposed Solution

The above problems made us abandon the initial idea of having a completely unrestricted read and write model and turn to developing a new simplified model to support having TLS run inside transactional code.

Proposed Solution Semantics

To use our simplified model, the programmer can hand-parallelize the program in coarse-grained threads, using transactions to ensure atomicity where needed, like he would do when using TM alone. Additionally, some of these transactions may have a specific structure that allows them to be further parallelized by using TLS. These transactions are split into three blocks:

- The starting block - In this block no writes to shared variables are allowed.
- The TLS block - In this block no writes to shared variables are allowed and no reads to shared variables read on the starting block are allowed. It is guaranteed that the transaction will not abort when inside the TLS block. TLS parallelized code can be run inside of this block, hence the name.
- The remainder of the transaction - In this part shared reads and writes are unrestricted. The transaction behaves like a regular transaction.

We use this structure to run the TLS parallelized code inside the TLS block. A transaction that does this is called a spec transaction. An example of such transaction can be seen in [Fig. 3]. In this example, the starting block starts at the beginning of the transaction (line 2) and ends in line 4. The TLS block starts in line 5 and ends in line 10. After line 10, we have the remainder of the transaction.

```
1:  function ProcessData(n, src, dst)  ▷ src and dst are shared variables
2:      atomic
3:          if not AlreadyProcessed(n) then
4:              SetInsideTLS(true)
5:              ▷ Parallelized by TLS
6:              for i ← 1, SIZE do
7:                  ▷ DoStuff only performs reads
8:                  localProcessedData[i] ← DoStuff(src[n][i])
9:              end for
10:             SetInsideTLS(false)
11:             Copy(localProcessedData, dst[n], SIZE)
12:             SetAlreadyProcessed(n, true)
13:         end if
14:     end atomic
15: end function
```

Figure 3: Example of spec transaction.

It is the responsibility of the programmer to choose where the TLS block starts and ends, and to ensure that the above restrictions are respected.

As explained in the above section, the only way to ensure that the TLS block never aborts is to never call any TM functions inside of it. Since shared writes are not allowed inside the TLS block, we only need to avoid calling the TM read function. To guarantee the correctness of the reads without using the TM read function, we need to ensure that:

- A shared read inside the TLS block only happens when no other running transaction is writing to the same memory location.
- A transaction cannot perform a write to a shared variable that was read inside the TLS block of another running transaction.

To enforce this conditions, our read function, when called inside the TLS block, will check if there is another transaction writing to the target variable. If there is, it will wait until the writing transaction ends. Otherwise, it will mark the location as being read by the current transaction and proceed with the read. Likewise, our write function will check if there are any other transactions that signaled they are reading the variable. If there are, the writing transactions will abort. Otherwise, it will mark the variable as being written by the current transaction and proceed with the write. Additionally, after the end of the transaction, each transaction will clear their mark on the locations they read inside the TLS block or wrote.

The restrictions imposed on the programmer and the use of locks detailed above allow us to guarantee that the code inside the TLS block reads correct values and never aborts.

In spite of having to follow this specific structure, we believe good results could be achieved. The programmer already had to parallelize the program in coarse-grained threads to take advantage of TM, and now further parallelism is possible by using TLS on suitable transactions.

Proposed Solution Implementation

Before we start explaining the implementation of our algorithm, we start with explaining some notation used throughout this section:

- TM_Read(x) - Function called to read a shared variable, using TM.
- TM_Write(x, val) - Function called to write to a shared variable, using TM. Writes *val* to *x*.
- TM_Start - Function called immediately after a transaction starts (at the beginning of the atomic block).
- TM_End - Function called after a transaction ends (either committing or aborting).

The previous functions are implemented by our system. Functions with the prefix _Original are the functions of the TM and TLS systems the programmer choose to use.

Global and Local Variables

Because we are using the existing TM and TLS systems like black-boxes, we cannot access any of its internal data. This means we need to keep track of the read and written memory locations ourselves. In order to ensure the semantics mentioned in the previous section, we need to keep track of the memory locations being written inside of running transactions. Additionally, we also need to keep track of the memory locations being read inside TLS blocks. To keep track of both, we use a global array of 64 bit integers. This array is called addr_counts. Each memory location is mapped to a position in this array (one array position can belong to more than one memory location). Each position is split in two: the least significant half is used for the TLS reads and the most significant half is used for TM writes. We use each bit to represent a running transaction. This means that each running transaction uses two bits in each array position. A transaction with id n will use bit n to signal that performed a TLS read

on the location and bit n + 32 to signal that performed a TM write.

These memory locations work like locks, keeping the TLS shared reads on hold and aborting the TM writes when it is not possible to gain access to the required address. These locks are released after the transaction ends.

There is only one transaction local variable. It is a boolean named inside_tls and its purpose is to tell if a transaction is inside the TLS block.

Performing Shared Reads

The code of the TM_Read function can be seen in [Fig. 4]. A shared variable read can happen either inside or outside the TLS block. When it happens outside the TLS block, no extra actions are needed, we just call the original TM function (line 3). This means there is no extra overhead on reads made by regular transactions. However, when inside the TLS block, we need to make sure that no other transaction is writing to the variable we are trying to read. In this case, we first compute the position of the address in the global_counts array (line 5). Then we check if this transaction already read this variable inside of its TLS block (lines 6-7). If it did not, we need to update the addr_counts position to signal other transactions that they cannot write to this address (lines 8-11). Otherwise, we know that addr_counts was already correctly updated and we can just return the read value (line 13).

```
1:  function TM_Read(addr)
2:      if not inside_tls then
3:          return TM_Read_Original(addr)
4:      end if
5:      h ← Hash(addr)
6:      tls_mask ← ShiftLeft(1, tx_id)
7:      if not (addr_counts[h] & tls_mask) then
8:          do
9:              counts ← addr_counts[h]
10:             other_writer ← (Highest32Bits(counts) ≠ 0)
11:         while other_writer or not CAS(addr_counts[h], counts, counts | tls_mask)
12:     end if
13:     return *addr
14: end function
```

Figure 4: Implementation of the TM_Read function.

Performing Shared Writes

The code of the TM_Write function can be seen in [Fig. 5]. When performing a write, we first compute the position of the address in the global_counts array (line 2). Then, we check if the current transaction already wrote to the address (lines 3-4). If it did, we can just call the TM write function (line 14). Otherwise, we need to update the addr_counts array (lines 6-12). The logic is similar to the case of a shared read inside the TLS block. Before updating addr_counts, we check if any running transactions, other than this, performed shared reads in their TLS block (lines 8-11). If they did, we abort the transaction (line 10), since we cannot write to variables being read by the TLS block of other transactions. Otherwise, addr_counts is updated (line 14).

```
1:  function TM_Write(addr, val)
2:      h ← Hash(addr)
3:      tm_mask ← ShiftLeft(1, tx_id + 32)
4:      if not (addr_counts[h] & tm_mask) then
5:          tls_mask ← ShiftLeft(1, tx_id)
6:          do
7:              counts ← addr_counts[h]
8:              tx_reading ← (Lowest32Bits(counts) ≠ 0)
9:              current_tx_read ← (Lowest32Bits(counts) = tls_mask)
10:             other_tls ← (tx_reading and not current_tx_read
11:             if other_tls then
12:                 TM_Abort_Original()
13:             end if
14:         while not CAS(addr_counts[h], counts, counts | tm_mask)
15:     end if
16:     TM_Write_Original(addr, val)
17: end function
```

Figure 5: Implementation of the TM_Write function.

Finishing a Transaction

The code of the TM_End function can be seen in [Fig. 6]. At transaction end (either after commit or abort), we need update the positions of the addr_counts array, removing the values we added before (lines 6-8). This is done by clearing the two bits that belong to the running transaction (line 7).

If the addr_counts array is very large, the transaction end routine can take a significant amount of time. To mitigate this problem, we can reduce the size of the addr_counts array. However, if the array is too small, more false-positive conflicts between TM and TLS will be detected.

4. Results

In this chapter we will present the results we obtained by our system. The main goal of this section is to answer the question: In which cases is it beneficial to add spec transactions to an existing TM application?

In order to answer these question, two different benchmarks were used to test our system. The first

```
1: function TM_End
2:     TM_End_Original()
3:     tm_mask ← ShiftLeft(1, tx_id + 32)
4:     tls_mask ← ShiftLeft(1, tx_id)
5:     both_mask ← (tm_mask | tls_mask)
6:     for i ← 0, ADDR_COUNTS_SIZE − 1 do
7:         addr_counts[i] ← addr_counts[i] & ∼ both_mask
8:     end for
9: end function
```

Figure 6: Implementation of the TM_End function.

one is an already existent benchmark, present in the STAMP suite [2] of benchmarks and it is called vacation. The second one is a custom benchmark made by us which uses a red-black tree as its main data structure. In Section Methodology we present the hardware used and how the benchmarks were run. Section Vacation details the vacation benchmark, how TLS was added and discusses the obtained results. Section Red-Black Tree Benchmark does the same for the red-black tree benchmark.

Methodology

The benchmarks were run on a machine with 4 AMD Opteron 6272 CPUs, making a total of 64 hardware threads. In order to obtain more accurate results, each benchmark was run three times and the results are the average of all three runs.

Each benchmark has four versions:

- Single-threaded version (sequential) - Completely sequential version without any TM or TLS instrumentation.
- TM parallelized version (TM-only) - Version parallelized using only TM. No TLS instrumentation.
- TLS parallelized version (TLS-only) - Version parallelized using only TLS. No TM instrumentation.
- TM and TLS parallelized version (TM-TLS) - TM is used for coarse-grained parallelization and our system is used to allow TLS to parallelize suitable transactions.

The blocks of code parallelized by TM in the TM-only version and in the TM-TLS version are the same. Similarly the blocks of code parallelized by TLS in the TLS-only version and in the TM-TLS version are also the same.

For each version we measured the time taken to run the application to completion with the mentioned parameters. Additionally, for the TM and TLS parallelized version, we also measured the number of transaction aborts that were caused by the introduction of our system. In the results shown, in the cases when there is only one TM thread, the TM parallelized version was used. Similarly, when there is only one TLS thread, the TLS parallelized version was used. Finally, the case with one TM and one TLS thread is the single-thread version, without any instrumentation. All other combinations use our system.

We tested the results with 2, 4 and 8 TLS threads. Due to the limitations in SpLip, it was not possible to use more than 8 threads. We tested the TM system with 2, 4, 8, 16, 32 and 64 threads. Combinations where the total count of threads exceeds the number of hardware threads available were ignored.

Vacation Benchmark

Vacation is an application that simulates a travel reservation system by implementing a transaction processing system powered by a non-distributed database. The main data structures used in this benchmark are red-black trees used to implement the tables that keep track of customers and travel items, which can be rooms, cars or flights. The tables for the travel items have relations with fields representing an unique ID, the reserved quantity, the available quantity and the price of each item. The customers table keeps track of the reservations made by each customer and of the total price of said reservations.

This benchmark consists in running several transactions in parallel, and each transaction can perform one of three different operations: to make a new reservation, to update the items to be reserved or to delete a reservation. Make a new reservation will check the price of N travel items and reserve some of them. Delete customer will pick a random customer and compute the cost of all the its reservations and remove them and the customer. Update the items to be reserved will add or remove N travel items.

There are several parameters that can be used to configure this benchmark:

- Number of queries per operation (N) - Number of queries each operation will perform. Larger numbers will create longer transactions.
- Percent of relations queries (Q) - Range of values from which each operation generates customer and travel items ids. Smaller values will create higher contention.
- Percent of user operations (U) - Controls the distribution of the operations. U% will be make reservation. Half of the remaining operations will be delete reservation and half will be update the items to be reserved.
- Number of possible relations (R) - The number of entries that the tables will have initially.

- Number of operations (T) - Total number of operations to be performed.

Adding TLS to Vacation
After looking at the implementation of the operations above, two of them have an inner loop: make reservation and update travel items. Because we do not allow writes to shared variables inside the TLS code, the only operation that we can apply TLS to is to make new reservation. This operation consists of two parts. The first part will go through N travel items, check their price and, if its larger than the previous price found for that type of item, stores its price and id locally. In other words, it will find the most expensive travel item for each type. The second part will reserve the travel item selected for each type. The first part of this transaction can be parallelized by TLS since it only writes to local variables. The writes to shared variables are all done on the second part.

The code that we want to parallelize is, essentially, finding out the items with the maximum price. However, searching for a maximum value using TLS has terrible performance, due to the fact that each time a new maximum value is found, it will be written to a variable shared by all TLS threads. This will cause many conflicts that will make the speculative threads abort. Since TLS only achieves good results when number of conflicts is very low (less than 1% [10]), this approach will have very poor performance. To solve this problem, we choose an alternative approach, instead of having a variable with the maximum price shared by all TLS threads, each TLS thread stores locally the maximum price it found and, after all threads finish, we compute the maximum value among all TLS threads.

Evaluation Results
To run this benchmark, the following parameter values were used:

- Number of queries per operation (N) - 8192
- Percent of relations queries (Q) - 90
- Percent of user operations (U) - 98
- Number of possible relations (R) - 16384
- Number of operations (T) - 4096

All the values were set as recommend by the benchmark authors for a low contention run, except for N. The recommend value for N was 2. The value of N controls the size of the loop used check travel items prices in the make reservation operation. As mentioned before, this is the only operation where TLS can be applied. This operation has a loop that goes through the travel items. In order to make this benchmark benefit from the use of TLS, this loop needs to be relatively large. This happens because if the loop is very small, the TLS overhead will be a lot larger than the gains obtained by running the loop in parallel.

To run these benchmarks, our system was configured to use 16 address bits.

We focused on studying the performance results of the algorithm on this benchmark by comparing them with the sequential, TM-only and TLS-only versions of the benchmark. All thread combinations were tested. After running the vacation benchmark with the needed changes to use our system we measured the execution times in [Fig. 7].

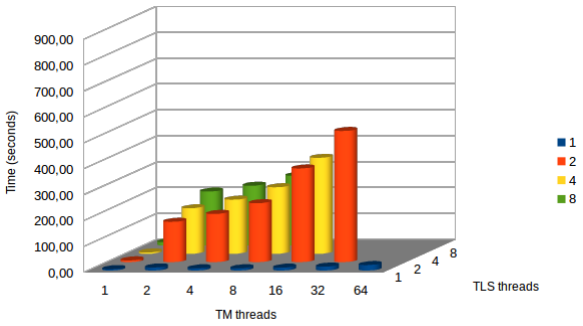

Figure 7: Execution time for vacation benchmark using our algorithm with 16 address bits.

As it can be seen from results, in this benchmark, the results are very poor. This is caused by the increase of N to very large values. Large values of N have adverse effects on the performance of the benchmark as a whole. A large increase of this value will create a very large transaction that will spend the majority of its time in the parallelized loop, reading shared variables. After the loop ends, there may be a write (if items to reserve were found) and the transaction may abort there. This is visible even in the TM-only version of the benchmark. The performance with multiple threads is worse than the single-thread version. Additionally, the mentioned loop will read the shared red-black trees that are shared among other operations. This will cause the other transactions running other operations concurrently to detect that the variables they are trying to write are being locked by the TLS code and cause an abort. The percentage of the total aborts that were caused by TLS block locking locations for reading is 99.9%.

From these results we can conclude that when the TLS code performs reads on a large percentage of the shared variables that are written by other transactions the performance tends to be very poor. This happens due to the fact that the TM writes will repeatedly cause a transaction abort while attempting to read shared data being read by the TLS par-

allelized code. If the write transactions were also long, the reverse situation would also happen: the transactions performing reads inside the TLS parallelized code would repeatedly have to wait for the writing transactions to finish. From these results, we can conclude that this system is not suitable in cases where there is a large percentage of shared data between TLS parallelized code and regular write transactions.

Red-Black Tree Benchmark

This is a custom benchmark made by us. This benchmark consists of an application that will cipher random blocks of data using the IDEA symmetric block cipher [9]. The data structures used by this benchmark consist of two arrays and a red-black tree. One array contains blocks of plain-text data to be ciphered and the other will store the ciphered data. The red-black tree keeps track of the indexes of the array blocks that are already ciphered.

The benchmark consists of running several transactions in parallel, each transaction performs one of three possible operations: add, remove or lookup. The add operation checks if a block is already ciphered (if the index is on the red-black tree) and, if it is not, ciphers it and adds its index to the red-black tree. Remove will remove a block from the red-black tree. Lookup checks if a block is ciphered and if it is, it will read the block and search for a specific string. The add operation uses the TLS to parallelize the cipher of the block.

There are several parameters that can be used to configure this benchmark:

- Number of operations (T) - Total number of operations to be performed.
- Percentage of lookup operations (L) - Controls the distribution of the operations. L% will be lookups. Half of the remaining operations will be adds and half will be removes. Higher values will increase the amount of read-only transactions.
- Number of blocks (S) - Total number of blocks on the arrays with plain-text data and ciphered data. Decreasing this value increases the probability of concurrent operations choosing the same block, therefore increasing contention.
- Rounds (R) - Controls the size of the blocks to be ciphered and the number of operations inside each iteration of the cipher algorithm. The size of each block is R*R/2. Increasing this value increases the time to run the cipher algorithm and therefore, increases the transaction size too.

Adding TLS to Red-Black Tree

The remove operation consists of removing a value from the red-back tree. As such, it is not a candidate for using TLS. In the add operation, there is a loop used to cipher the data and the lookup operation could be manually implemented with a loop to search for the string. This makes them good candidates to be parallelized by TLS. However, the size of lookup operation is too small to get performance benefits of applying TLS. In the add operation, the ciphering of the data block is parallelized by TLS. Since we cannot perform shared writes inside the TLS code, the output of the cipher is first written to a local array. After the TLS execution ends, the local array is copied to the global array using TM. Also notice that the global array with data to be ciphered is only written at the beginning of the benchmark, before the transactions start. After that, it is only read. This means that in spite of being shared among threads, this array needs no synchronization, since it is only read concurrently, never written. Because the ciphering code does not read any variables written by other transactions, the TLS block of this transaction will never cause another transaction to abort.

Evaluation Results

To run this benchmark, the following parameter values were used:

- Number of operations (T) - 128
- Percentage of lookup operations (L) - 20
- Number of blocks (S) - 256
- Rounds (R) - 1024

To run these benchmarks, our system was configured to use 16 address bits.

In this section, we focus on studying the performance results of our algorithm on this benchmark by comparing them with the sequential, TM-only and TLS-only versions of the benchmark. All thread combinations were tested. After running the red-black tree benchmark with the needed changes to use our system we measured the execution times in [Fig. 8].

In this benchmark, we were able to achieve positive results. The results show that adding TLS to the version already parallelized by TM increases performance in every case tested and adding TM to the version parallelized by TLS increases performance too. Furthermore, increasing the number of TM or TLS threads almost always results in increased performance too.

5. Conclusions

We developed a system that allows a programmer to combine different TM and TLS algorithms with

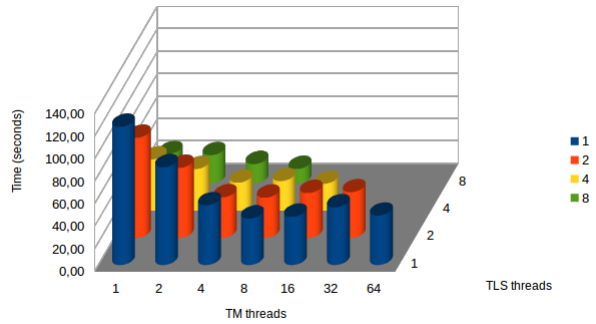

Figure 8: Execution time for red-black tree benchmark using our algorithm with 16 address bits.

little effort. In this article, we discuss the details of our approach and evaluated it to identify the cases where our approach performs best.

Future Work

Our results show that our system performs well when there are no conflicts between TLS reads and TM writes, and poorly when there are many conflicts. It would be useful to develop a new benchmark to determine what is the percentage of conflicts that causes or system to stop improving performance.

Although the goal of our work is the development of a system that allows TM and TLS algorithms to be joined together, the system we developed seems to solve a more generic problem than that. The semantics of the new spec transactions we developed provide the programmer with a guarantee that there is a section when the transaction will not abort. While we are using this section to run TLS parallelized code, there may be other situations where such semantics might be useful. The identification of instances where such semantics might be worthy of more research effort.

References

[1] J. a. Barreto, A. Dragojevic, P. Ferreira, R. Filipe, and R. Guerraoui. Unifying thread-level speculation and transactional memory. In *Proceedings of the 13th International Middleware Conference*, Middleware '12, pages 187–207, New York, NY, USA, 2012. Springer-Verlag New York, Inc.

[2] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[3] N. Carvalho, J. a. Cachopo, L. Rodrigues, and A. R. Silva. Versioned transactional shared memory for the fénixedu web application. In *Proceedings of the 2nd workshop on Dependable distributed data management*, WDDDM '08, pages 15–18, New York, NY, USA, 2008. ACM.

[4] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. *SIGPLAN Not.*, 33(11):58–69, Oct. 1998.

[5] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.

[6] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2002.

[7] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[8] E. Koskinen and M. Herlihy. Dreadlocks: efficient deadlock detection. In F. M. auf der Heide and N. Shavit, editors, *SPAA*, pages 297–303. ACM, 2008.

[9] X. Lai and J. L. Massey. A proposal for a new block encryption standard. pages 389–404. Springer-Verlag, 1991.

[10] C. E. Oancea, A. Mycroft, and T. Harris. A lightweight in-place implementation for software thread-level speculation. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 223–232, New York, NY, USA, 2009. ACM.

[11] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *IEEE PACT*, pages 303–313. IEEE Computer Society, 1999.

[12] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *25 Years ISCA: Retrospectives and Reprints*, pages 521–532, 1998.

[13] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero. Atomic quake: using transactional memory in an interactive multiplayer game server. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 25–34, New York, NY, USA, 2009. ACM.