TÉCNICO LISBOA

specSTM: Software Transactional Memory with Thread-Level Speculation Support

Daniel Filipe Soares Pinto

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor(s): João Pedro Faria Mendonça Barreto
Supervisor(s): Paolo Romano

Examination Committee

Chairperson:	Professor Full Name
Supervisor:	João Pedro Faria Mendonça Barreto
Supervisor:	Paolo Romano
Member of the Committee:	Professor Full Name

May 2015

Resumo

Com a proliferação dos processadores multi-core, torna-se cada vez mais importante para os programadores escreverem programas que aproveitem a existência de múltiplos cores. No entanto, escrever estes programas é uma tarefa difícil.

Para simplificar a criação de programas multi-threaded, várias abordagens existem. Duas dessas abordagens são a Memoria Transacional (MT) e a Thread-Level Speculation (TLS). Ambas as abordagens tem vantagens e desvantagens, ao nível da facilidade de utilização e do desempenho obtido. Apesar da TLS ser fácil de utilizar, só consegue obter melhorias de performance em cenários muito específicos. A MT é mais difícil de utilizar, no entanto, é mais fácil conseguir melhorias de performance.

Em vez de escolher entre MT e TLS, nós acreditamos que combinar TM com TLS num único sistema resultará em mais parallelismo. Para utilizar a nossa abordagem, o programador paraleliza o seu programa em transações de grau grosso, usando MT. Adicionalmente, se algumas transações forem apropriadas, a TLS pode ser utilizada para paralelizar ainda mais essas transações. Além disso, o nosso sistema permite ao programador trocar os algoritmos TM e TLS utilizados, permitindo-lhe facilmente tirar partido dos novos desenvolvimentos nas áreas de investigação de TM e TLS.

O nosso sistema foi testado com dois benchmarks que nos permitiram identificar situações onde o nosso sistema tem um desempenho melhor que a TLS ou a TM por si só.

Palavras-chave: memória transacional, thread-level speculation, computação paralela, sistemas muti-core

Abstract

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. Two of them are Transactional Memory (TM) and Thread-Level Speculation (TLS). Both approaches have advantages and disadvantages, regarding of ease of use and obtained performance. While TLS is simple to use, it only achieves performance benefits in very specific scenarios. TM is more difficult to use, however it is easier to achieve performance improvements with it.

Instead of choosing between TM or TLS, we believe that combining TM and TLS in a single system could result in increased parallelism. To use our approach, the programmer would hand-parallelize the program in coarse-grained threads, using TM. Additionally, if some of the transactions where suitable, TLS can be used to further parallelized them. Furthermore, our system allows the programmer to switch the TM and TLS algorithms used, allowing him to easily reap the benefits of new TM and TLS research.

Our system was tested with two benchmarks which allowed us to identify situations where we achieved better results than either TM or TLS alone.

Keywords: transactional memory, thread level speculation, parallel computing, multi-core systems

Contents

List of Tables

List of Figures

Glossary

CAS Compare And Swap.

CPU Central Processing Unit.

HLE Hardware Lock Elision.

HTLS Hardware Thread-Level Speculation.

HTM Hardware Transactional Memory.

IDEA International Data Encryption Algorithm.

RAW Read-After-Write.

RTM Restricted Transactional Memory.

STLS Software Thread-Level Speculation.

STM Software Transactional Memory.

TLS Thread-Level Speculation.

TM Transactional Memory.

TSX Transactional Synchronization Extensions.

WAR Write-After-Read.

WAW Write-After-Write.

Chapter 1

Introduction

Since 2005, CPU manufacturers have been releasing multi-core processors for the personal computer market. Since then, CPUs have been getting more and more cores and it is now affordable to acquire personal computer with four or more cores. Even mobile phones now have multi-core CPUs [7]. This trend tells us that the number of cores will keep increasing and mainstream machines may soon have tens or hundreds of cores [21]. In order to obtain a significant performance increase from the increasing number of cores, programmers have to create concurrent programs [41]. To make a program concurrent, it is first necessary to identify the parts of the code that can be forked into threads and then identify the sections of code that must run in an atomic way. However, this is a difficult task and requires a detailed understanding of the programs semantics.

The traditional approach to assure atomicity is to use lock-based synchronization [18], but locks are difficult to use, particularly in large software applications. Good programming practices dictate that programmers should be able use a function without knowing any details of its internal implementation. However, locks prevent this from happening. To illustrate why, consider the code in [Fig. 1.1]. If the transfer function is called from two different threads with the source and destination accounts reversed, both threads could acquire the first lock and both would then deadlock when trying to acquire the second lock [Fig. 1.2]. In a large application, composed by many individual pieces of code, ensuring that this does not happen would be a very challenging task.

Additionally, the non-composability of locks hinders the development of modular code. An example of this can be seen in [Fig. 1.3]. In the example, the Transfer function is incorrect despite the fact is composed by functions that are are correctly implemented. In this example, the CheckBalance function might return wrong results if it is run between the withdraw and the deposit operation of the Transfer function.

To mitigate the problems of lock-based synchronization, other paradigms were introduced. One of these paradigms is Transactional Memory (TM) [17]. In order to use TM, the programmer has to identify the blocks of the program that must run atomically and enclose them in atomic blocks. The code enclosed in an atomic block will run atomically because the TM system will do the needed work to ensure it, therefore alleviating the programmer from nontrivial concurrency issues. without the programmer

```
1: function TRANSFER(src, dst, amount)        ▷ Transfers amount from account src to account dst
2:     lock(src)
3:     lock(dst)
4:     try
5:         if src.balance ≥ amount then
6:             src.balance ← src.balance − amount
7:             dst.balance ← dst.balance + amount
8:         else
9:             throw InsufficientFundsException()
10:        end if
11:    finally
12:        unlock(dst)
13:        unlock(src)
14:    end try
15: end function
```

Figure 1.1: Example of synchronization using locks

Thread 1	Thread 2
Transfer(acc1, acc2, 100)	Transfer(acc2, acc1, 50)
lock(acc1)	lock(acc2)
lock(acc2)	lock(acc1)

Figure 1.2: Example of how the code in [Fig. 1.1] may cause a deadlock

having to know about it. An example of the previous code parallelized using TM is shown in [Fig. 1.4]. This approach is much easier to use because the programmer does not need to use a lock for each shared variable and reason about the order of the locks, therefore preventing deadlocks. It also allows programmers to use this function without needing to know about the synchronization mechanisms it uses, allowing for more composable functions and more modular code.

Although TM makes it easier for the programmer to parallelize the program, it still has significant limitations. To fork a new thread the programmer must have a deep understanding of the program semantics and make sure that the forked blocks are commutative for every possible execution order. If this does not happen, the program will behave incorrectly. This means that the common programmer will opt for the safe approach of not forking a thread that may be correct to assure that the program still works. Additionally, when forking a new thread the programmer has to consider that the additional overhead created by the new thread may not offset the gains obtained by the new thread. This can lead to an overestimation of the overheads and make the programmer follow the conservative approach of only forking new threads for large blocks of code. Furthermore, most current TM solutions do not support parallel nesting [1], reducing even further the opportunities for the programmer to exploit the benefits of fine-grained parallelism within a transaction. Because of these limitations, most TM programs are still organized as a small number of coarse-grained threads. This is evidenced by some representative TM-based applications [43, 4].

Another paradigm is Thread Level Speculation (TLS) [39, 13] which, may or may not require changes to the program, depending on the TLS implementation. TLS parallelizes the program by attempting to run several tasks (parts of the sequential program) speculatively in parallel. When a speculative task performs an operation that violates the serial program order, the changes made by the task are discarded

```
1: function DEPOSIT(acc, amount)
2:     lock(acc)
3:     acc.balance ← acc.balance + amount
4:     unlock(acc)
5: end function
6: function WITHDRAW(acc, amount)
7:     lock(acc)
8:     if src.balance < amount then
9:         unlock(acc)
10:        throw InsufficientFundsException()
11:    end if
12:    acc.balance ← acc.balance − amount
13:    unlock(acc)
14: end function
15: function TRANSFER(src, dst, amount)        ▷ Transfers amount from account src to account dst
16:    withdraw(src, amount)
17:    deposit(dst, amount)
18: end function
19: function CHECKBALANCE(acc)
20:    return acc.balance
21: end function
```

Figure 1.3: Example of the non-composability of locks: the Transfer function is incorrect even though Withdraw and Deposit are correct

```
1: function TRANSFER(src, dst, amount)         ▷ Transfers amount from account src to account dst
2:     atomic
3:         if src.balance ≥ amount then
4:             src.balance ← src.balance − amount
5:             dst.balance ← dst.balance + amount
6:         else
7:             throw InsufficientFundsException()
8:         end if
9:     end atomic
10: end function
```

Figure 1.4: Example of synchronization using transactional memory

and the is run again. A common use of a TLS system is parallelizing loops. When parallelizing a loop, the TLS would run one or more loop iterations per thread. In this case, examples of violations would be an iteration reading or writing a variable already written by a future iteration or an iteration writing data that a future thread already read. Although TLS systems can achieve good speed-ups when the number of conflicts is very low (usually less than 1% [33]), most programs that do not belong to the category of embarrassingly parallel problems will cause many conflicts. These conflicts will restrict the number of tasks that the TLS system can parallelize effectively [34] and thus limit the performance of the TLS system.

Instead of choosing between TM or TLS, we can combine TM and TLS in a single system, resulting in increased parallelism. This approach has been supported by TLSTM [2]. To use TLSTM, the programmer will hand-parallelize the program in coarse-grained threads (user-threads), using transactions to ensure atomicity where needed. Then the runtime will further split the transactions into multiple smaller tasks, transforming the transaction into a sequence of one or more tasks [Fig. 1.5]. The code outside

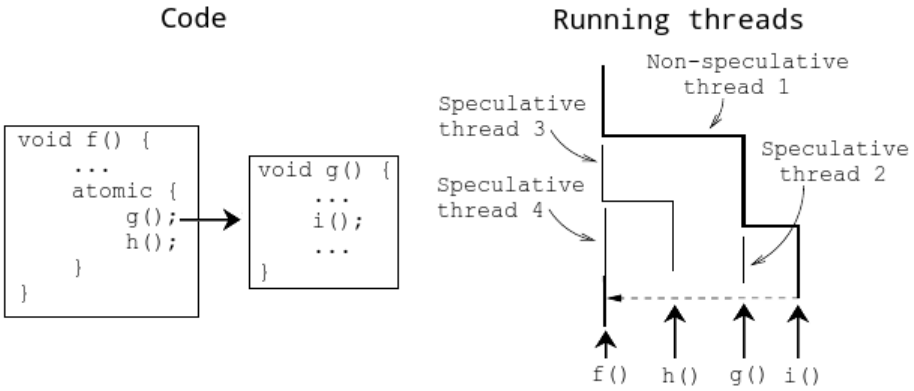

Figure 1.5: Example of task division. Horizontal lines in a thread represent function calls and vertical lines represent the execution of a function.

these transactions can also be split in a similar fashion. The tasks within a transaction (or outside any transaction) would then run speculatively in parallel allowing for further parallelism if no conflicts are detected. A conflict happens when the speculative execution of a task is inconsistent with the expected outcome of the serial execution of the user-thread the task belongs to. When a conflict happens, the changes made by the most speculative task will be discarded. In order to further increase parallelism, we can even be more optimistic and start executing future transactions of the same user-thread even when the current transaction is still not finished. If both the current and the future transaction commit, further parallelism is achieved.

Despite the positive results, TLSTM still has a significant limitation. It is not possible to use new advances in the TM and TLS fields easily. If new and more efficient TM or TLS algorithms are developed, it is not possible to integrate them in TLSTM. In order to take advantage of the performance benefits provided by new research, it is currently necessary to develop a system like TLSTM for each new TM or TLS algorithm. We believe it would be beneficial for the programmers to have a system that allows them to easily reap the benefits of new TM and TLS systems. However, joining TM and TLS systems together is not easy. TLS runs under the assumption that is running on a single-threaded program. This assumption no longer holds when we add TM. On the other hand, TM assumes that every shared read and write goes through the TM functions. This assumption also fails when TLS is added to the program. Our thesis is that combining off-the-shelf TLS and TM algorithms can be done with a positive outcome. The main goal of our work is to allow the programmer to easily choose among different TM and TLS algorithms when building their program. This allows for easy replacement of the initial algorithms with improved algorithms, without the need to modify or understand such algorithms. We developed a system that allows a programmer to pick existing TM and TLS algorithms and combine them together without modifying either. To use our approach, the programmer can parallelize the program using a model similar to TM. Additionally, a new kind of transaction, called spec transaction is available to the programmer. The spec transactions are special transactions that allow the programmer to run code parallelized by TLS inside of the transaction. To ensure the correctness of the program, these transactions have some restrictions. The TLS parallelized code cannot perform any writes to shared variables and has to be run before the transaction makes any write to shared variables. Additionally, shared variables read before

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

Figure 1.6: Example of spec transaction

the TLS parallelized code cannot be read inside of it. After the TLS parallelized code ends, there are no more restrictions and the remainder of the transaction can be written as a regular transaction. On transactions with a suitable structure, spec transactions could be used, allowing for more parallelism than using TM alone. An example of a spec transaction can be seen on [Fig. 1.6]

We tested our system using SpLip [33] and RSTM [40] on the vacation benchmark from STAMP suite [3] and on new benchmark developed by us. Our results show that our system can achieve better results than using either TM or TLS alone.

This remainder of this dissertation is organized as follows:

- *Related Work* - Chapter 2 details some aspects of existing solutions for parallelize programs. Namely Automatic Parallelization, TM, TLS and previous approaches to join TM and TLS.
- *Architecture* - Chapter 3 provides some insight on how our system can be used and gives a general overview of how and why it works.
- *Implementation* - Chapter 4 goes more in depth on the inner workings of our system than the previous chapter and presents three different implementations for our system.
- *Evaluation* - Chapter 5 presents the benchmarks used to evaluate our system and the results obtained.
- *Conclusions* - Chapter 6 summarizes the work described in this dissertation, the results achieved, and what are its main contributions. It also presents some possible future improvements to our proposed solution.

Chapter 2

Related Work

When trying to parallelize a program, there are several approaches available. These approaches vary in easiness for the programmer and obtained performance. Some approaches are completely automatic requiring no programming effort, while others require the programmer to parallelize everything manually and others try to mix both concepts. Typically, there is a trade-off between programmers effort and performance. The approaches that are easier for the programmer yield poorer parallelism than the approaches that require more of the programmers time and expertise.

In this section, we describe some of the available approaches. We start by the fully automatic approaches such as automatic parallelization done by the compiler and Thread-Level Speculation. After that we describe Transactional Memory, a manual approach. Finally, we describe the mixed approaches.

Automatic Parallelization

One of the possible approaches to automatically parallelize a program is the use of dependence analysis to identify loop iterations that can run independently in parallel. Although this can achieve positive results in some scientific applications [5], this does not usually happen with many non-trivial applications [33]. One of the techniques used to automatically parallelize loops is DOALL [27]. A loop can be parallelized using DOALL if all of its iterations, in any given invocation, can be executed concurrently. While DOALL generally scales well with the number of iterations of the loop, its applicability is limited by the presence of loop-carried dependences. Unfortunately, DOALL-style parallelism has limited applicability because of inter-iteration dependences. These dependences are common in most application codes outside the scientific and data-parallel arenas. Even in the domain of scientific applications, it is common for DOALL parallelism not to be directly available without employing other transformations [27].

Another approach is DOACROSS [8] which, contrary to DOALL, can handle loops that have dependences. When using DOACROSS, each iteration is assigned to a processor for execution. Any inter-iteration dependences that may happen are resolved using explicit fine-grained processor synchronization. Even though DOACROSS attempts to handle dependences it is still restricted to loops with very analyzable, array-based memory accesses, and does not handle arbitrary control flow [37].

Another parallelization technique that parallelizes loops in spite of the presence of dependences is decoupled software pipelining (DSWP) [37]. DSWP is able to parallelize loops with irregular, pointer-based memory accesses and arbitrary control flow which allows it to parallelize general-purpose applications [37]. DSWP parallelizes a loop by partitioning the instructions of a loop among a sequence of loops. Then, it will execute the new loops concurrently on different threads, with dependences among them flowing in a single direction, thus forming a pipeline of threads. Because the threads form a pipeline, DSWP is not affected by increases in the inter-processor communication latency. However, the scalability of DSWP is limited by the size of the loop body and the number of recurrences it contains, which are usually smaller than the loop iteration count [37].

Even when dependence analysis fails to prove the absence of dependences, parallelism may still be possible with more optimistic techniques such as TLS.

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.

Unlike other automatic parallelization approaches, TLS does not require that the threads created are proved to be independent. Instead, threads are created speculatively to run blocks of code that are likely independent. However, there is no guarantee that a conflict will not happen. If a a conflict happens, the system must detect it and resolve it at run-time and ensure that the end result is the same as if the code had run in the original sequential order. This is generally done by restarting the threads that caused conflicts. 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 program, due to the cost of rollbacks, conflicts should still be avoided. Therefore, it is important to decide carefully when to spawn more threads. In order to reduce conflicts and improve parallelization, several techniques were developed to decide when to spawn more threads.

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

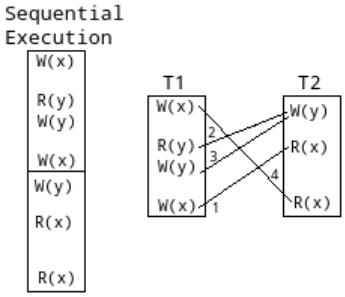

Figure 2.1: Types of possible dependences. R(x) means read from memory location x and W(x) means write to memory location y. (1) and (4) are flow dependences, (3) is an output dependence and (2) is an anti-dependence

```
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 2.2: Example of code parallelized using loop-level speculation

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. 2.1]. 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 [16].

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. 2.2]. A system that uses this approach is SpLip [33].

A different approach is the use of procedure fall-through speculation [42]. When using procedure fall-through speculation, a new thread is spawned when the code reaches a function call. The original thread will continue to execute the called function, while a new thread is created for execution of the remaining code, i.e. the code after the function call. An example of how this division could be made is shown in [Fig. 2.3]. Procedure fall-through speculation provides some advantages over loop-level speculation. Control dependence violations are, usually, not a problem, because a function will almost always return and the code after the function will be executed. Also, functions should access mostly local data, minimizing inter-thread dependences, except for the return value which is a common dependency.

This approach may be combined with return value prediction [34]. Functions are called in parallel with the code after the function which runs speculatively. Because the return value of a function is

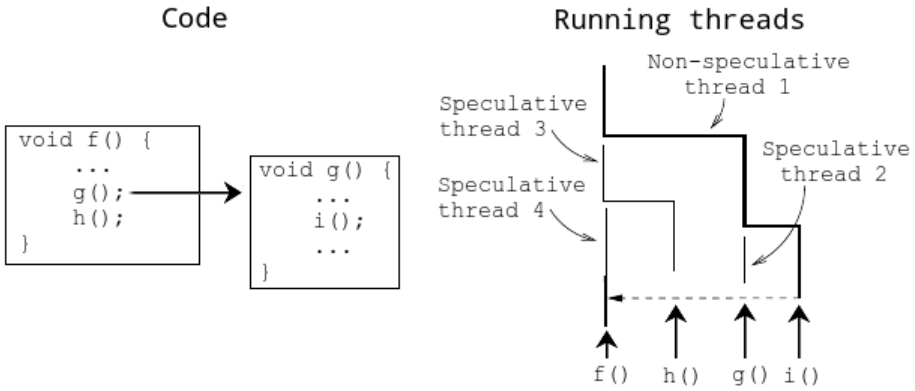

Figure 2.3: Example of code parallelized using procedure fall-through speculation

typically used immediately after the function call, procedure return value prediction is applied. The prediction is made by keeping a cache of the values returned by the function in the past. These values are used to execute the code after the function. In case the return value proves to be wrong, the speculative thread is aborted an the code is re-executed with the correct value.

Another approach is MEM-slicing [6]. MEM-slicing uses memory accesses to decide when to spawn a new thread. After a minimum number of instructions being executed, when a memory access occurs, a new thread will be spawn.

One more approach is the use of profile-based analysis to spawn new threads [31]. This approach uses three criteria to decide when to spawn a thread: thread size, few data dependences between threads and few control dependence violations between threads.

SpLip

SpLip [33] is a state of the art TLS system that uses loop-level speculation. SpLip shows positive results achieving, on average, 77% of the speed-up of the hand-parellelized versions of tested bechmarks. These results are achieved by using an algorithm that allows SpLip to perform in-place writes and avoid buffering of speculatively written values. In addition to this, it also avoids the need for a serial commit phase and the use of expensive operations such as CAS or memory fences.

Data Structures

SpLip makes use of several data structures internally, both to keep track of the versions of the addresses written to and to restore the overwritten values in case of an abort. In order to keep track of the versions of memory addresses, two arrays are used: LdVct and StVct. Each position of these arrays matches a set of memory addresses and their value holds the maximal interaction id that read or wrote to the matching addresses. To store the previous versions of overwritten memory locations, a shadow buffer (ShBuff) is used. The shadow buffer is two-dimensional array with P lines and W columns, where P is the number of processors and W is the maximum number of writes allowed per iteration. Each position of this buffer holds three values: the address being written, the previous value and a timestamp. There is also an additional array called StampVct that keeps the current timestamp for each set of each memory

```
1: function SPECLOAD(addr, itNum)
2:     atomic
3:         h ← hash(addr)
4:         if LdVct[h] < itNum then
5:             LdVct[h] ← itNum
6:         end if
7:         readValue ← *addr
8:         if StVct[h] ≤ itNum then
9:             return readValue
10:        else
11:            throw DepEx(itNum - 1)
12:        end if
13:    end atomic
14: end function
```

```
1: function SPECSTORE(addr, newValue, itNum)
2:     atomic
3:         h ← hash(addr)
4:         storeIter ← StVct[h]
5:         if storeIter > itNum then
6:             throw DepEx(itNum - 1)
7:         end if
8:         StVct[h] ← itNum
9:         save(addr, *addr, StampVct[h] + +)
10:        *addr ← newValue
11:        loadIter ← LdVct[h]
12:        if loadIter > itNum then
13:            throw DepEx(itNum - 1)
14:        end if
15:    end atomic
16: end function
```

Figure 2.4: Speculative load and store operations implemented by SpLip, assuming atomicity.

addresses. Additionally, there are also two more arrays, SyncR and SyncW that are used to avoid the need to perform CAS operations.

Load and Store operations

The initial implementation of the load and store operations assumes they execute atomically [Fig. 2.4]. In the following sections, we will explain the non-atomic versions.

Both SpecLoad and SpecStore start by computing the index of StVct and LdVct corresponding to the address being read and store it in variable h. The values stored in LdVct and StVct represent the number of the highest iteration that has read/written to the addresses that map into each index. These values increase monotonically in time (these is guaranteed by lines 4 and 5 of specLoad and specStore, respectively).

A speculative load succeeds if there was no previous speculative writes to the same address made by later iterations. This check is made in line 8. If the check fails, the current iteration is aborted and a DecEx exception in thrown with the argument itNum -1, which tells the framework that the highest correct iteration is itNum - 1.

A speculative store succeeds only if there has been no speculative accesses to the target memory

location by later iterations. The check for a later iteration write is made in line 5 and the check for a later iteration read is made in line 12. As with specLoad, any failure will throw and exception and signal the framework that the highest correct iteration is itNum - 1. When successful, specStore stores the previous value of the address in ShBuff along with the current timestamp (line 9) and updates the value in-place (line 10).

Rollback

This section shows the algorithm used in SpLip to run the speculative threads [Fig. 2.5]. When a speculative thread first starts, it will receive a new iteration id to execute and it will start by serially running the code that would yield too many dependency violations and the code that contains un-rollable operations (line 4). After that, the the induction variables whose value can be computed knowing only the current loop iteration are computed (line 5). The loop is unrolled U times to achieve the desired granularity (line 6).

When a dependency-violation, the end of the loop, or any other exception is caught, the thread will wait until it becomes the master thread (line 13). Then, it will kill its successors (line 14) and start the rollback procedure (line 15). The master thread is the thread executing the iteration with the lowest number. It is guaranteed that one of the threads that caused the violation will become master since its predecessors will eventually finish their iterations.

When the rollback procedure is called, the load and store vectors are cleared and the shadow buffer is used to restore the written memory address to the values they had on the highest correct iteration. The rollback algorithm has a complexity of O(W * C * log(W * C)) where W represents the maximum number of writes per iteration and C is the maximum number of concurrently running iterations.

Load and Store - non-locking implementation

In the previous sections, the presented algorithm assumed that specLoad and specStore run atomically. In this section, we will show how to make these methods concurrent without using CAS operations. In order to ensure the correctness of the algorithm, the following guarantees need to be met: (i) when there are several concurrent violations, the lowest numbered iteration that may have been involved in a dependency is detected and (ii) even when concurrent writes to the same address have the same timestamp the rollback procedure is still correct.

We start by analyzing the cases where running specLoad and specStore concurrently may lead to a violation of the sequential semantics. There are two such cases. The first case happens when specStore reaches line 11. There may be a specLoad call accessing the same index h in an inconsistent state: the value of LdVct[h] has been updated but the memory has not yet been read (specLoad execution is between line 5 and 7). When this happens, if the condition in line 12 of specStore holds, specStore will assume that the load read a value written by an earlier iteration. It will then signal a dependency violation, setting the highest correct iteration as itNum - 1. Because the specLoad itNum is necessarily greater than the specStore itNum - 1, the specLoad thread will be rolled-back, ensuring correctness.

```
1: function SPECTHREAD_RUN(void)
2:     while true do
3:         try
4:             id ← acqAndSerialExec()
5:             setIndVars()
6:             for i ← 1, U do
7:                 if isEndOfLoop() then
8:                     throw EndOfLoopExec()
9:                 end if
10:                iterationBody()
11:            end for
12:        catch
13:            waitBecomeMaster()
14:            killSpecThreads()
15:            rollbackProc(id − 1)
16:            if not specEnded() then
17:                respawnThreads()
18:            else
19:                return
20:            end if
21:        end try
22:    end while
23: end function
```

Figure 2.5: SpLips speculative thread layout.

The second case happens when a specLoad finds a specStore accessing the same index h in an inconsistent state: the thread running specStore (currentThread), updates StVct[h] (line 8) but the value read by the thread running specLoad (loadThread) in line 8 belongs to another earlier write made by another thread (writeThread). In this case, there are three possible scenarios:

- When the currentThread iteration is earlier than the writeThread iteration, the sequential order of writes was not respected. Because of that, the currentThread will detect the violation. When the loadThread iteration is earlier than the currentThread iteration, the currentThread will assume that value was overwritten and it will detect a violation. In both cases correctness is ensured.
- When the currentThread iteration is the same as writeThread, loadThread will consider the state consistent.
- Lastly, in the case where currentThread iteration is equal to StVct[h] and later than writeThread there are three possible scenarios:
  - if currentThread iteration is later than loadThread then loadThread will detect the violation in line 11.
  - if currentThread iteration is earlier than loadThread then, because of LdVct being monotonically increased, specStore will detect a violation in line 13 because currentThread $<$ LdVct[h].
  - the case where currentThread iteration is the same as loadThread is impossible because the same thread cannot be inside a load and a store at the same time.

  All tree scenarios ensure correctness.

```
1: function SPECLOAD(addr, itNum, thId)
2:     h ← hash(addr)
3:     SyncW[h] ← thId
4:     if LdVct[h] < itNum then
5:         LdVct[h] ← itNum
6:     end if
7:     if SyncW[h] ≠ thId then
8:         SyncR[h] ← itNum + P                    ▷ P = number of processors
9:     end if
10:    readValue ← *addr
11:    if StVct[h] ≤ itNum then
12:        return readValue
13:    else
14:        throw DepEx(itNum - 1)
15:    end if
16: end function
1: function SPECSTORE(addr, newValue, itNum, thId)
2:     h ← hash(addr)
3:     SyncW[h] ← thId
4:     storeIter ← StVct[h]
5:     if storeIter > itNum then
6:         throw DepEx(itNum - 1)
7:     end if
8:     StVct[h] ← itNum
9:     if SyncW[h] ≠ thId then
10:        throw DepEx(itNum - 1)
11:    end if
12:    save(addr, *addr, StampVct[h] + +)
13:    *addr ← newValue
14:    loadIter ← max(LdVct[h], SyncR[h])
15:    if (loadIter > itNum) or (StVect[h] ≠ itNum) then
16:        throw DepEx(itNum - 1)
17:    end if
18: end function
```

Figure 2.6: Speculative load and store operations implemented by SpLip.

Next, we will show the case of having two concurrent specLoads. In these case correctness is not ensured and changes to the initial algorithm were needed. These changes can be seen in [Fig. 2.6]. In order to understand how running two specLoads concurrently leads to wrong behavior, consider two threads t_0 and t_2, concurrently executing lines 4 to 9 (of [Fig. 2.4]) with the same addr. When t_0 is the last to be recorded on LdVct[i], LdVct will stop being monotonically increased and a later write made by a thread t_1 may fail to detect a RAW violation with t_2. To solve this problem, a value that is unique per-thread (thId) is stored in position h of the SyncW array (line 3). After the load, the value of SyncW[h] is checked and, if it has changed, a conflict is detected and SyncR[h] is assigned the value itNum + P (lines 7 and 8), where P is the number of concurrent threads. This value was chosen because it is impossible for any other thread to have an itNum greater or equal than itNum + P. The specStore function was also changed by adding a check for read conflicts in lines 14 and 15. These changes guarantee the monotonicity of LdVct.

The remaining problematic case is the case of concurrent specStores. The specStore method has a similar problem to the loadStore method with monotonicity of the StVct. The problem is solved with a

similar technique. In line 8, StVct[h] gets the value of itNum and, on line 15, it is checked to see if the value remains the same. When the values do not match a conflict is detected and an exception is thrown to signal the need to rollback. Although unlikely, it is possible that the lowest iteration involved in a race fails to detect it and then finishes the write operation and starts a new iteration. If that happens before another of the conflicting write initiates the rollback procedure, we will get incorrect results. To prevent this problem, SyncW[h] is assigned an unique thread value in line 3, like it was done with specLoad. This value is later checked in line 9 and a rollback is signaled if the values do not match. Without this modification, one of the rows in ShBuff, used for rolling back the changes, might have been overwritten. These changes fix the problem by ensuring that either the write operation causes a rollback or that lines 8 to 15 are run in mutual exclusion.

Finally, if line 12 is run concurrently, several writes to the same address may have the same timestamp. However, this does not cause a problem because these writes will have the same values since the address is read before the timestamp is incremented. As such, the rollback procedure may choose any of the values with conflicting timestamps and attain the same final result.

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 [15]. In a TM system the outside observer would be the other transactions executing in parallel.

The concept of transaction has its roots in database systems. Database transactions have four properties, known as the ACID properties. These are:

- Atomicity - either all actions inside a transaction complete successfully and the transaction commits or the transaction aborts and none of the effects produced by the actions inside the transaction are observable.
- Consistency - the meaning of consistency depends on the application semantics but typically is a set of invariants on data structures that must be respected. This means that a transaction will change the database from one consistent state to another.
- Isolation - transactions must not interfere with other running transactions, even when running in parallel.
- Durability - once a transaction commits, its result is permanently stored.

These properties also exist in transactional memory, except for durability which does not apply because memory is usually volatile.

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. Depending on the TM system, the programmer may have to restart the aborted transactions manually or the TM system can do it automatically. When a transaction aborts, it is due to a conflict with another transaction.

Ensuring correctness

Despite the fact that transactions are running concurrently, TM systems must run them while ensuring that the execution semantics of the program are respected. In order to ensure this, several correctness criteria were developed.

One of these criteria is linearizability [20]. Linearizability means that every transaction should appear as if it took place at some single, unique point in time during its lifespan. However, this is not enough to ensure correctness because a transaction is not a black box operation on some complex shared object but an internal part of an application: the result of every operation performed inside a transaction is important and accessible to a user. But linearizability only accounts for the whole transaction, it does not determine what should happen to the operations inside the transaction.

Another criterion is serializability [35], which states that the result of a history of transactions is serializable if all committed transactions in that history receive the same responses as if they were executed in some serial order. However, serializability does not provide any guarantee about accesses made by live (or aborted) transactions.

To solve this problem, another criterion was created, the opacity correctness criterion [12]. The opacity correctness criterion states that:

- All operations performed by every committed transaction appear as if they happened at some single, indivisible point during the transaction lifetime.
- No operation performed by any aborted transaction is ever visible to other transactions (including live ones).
- Every transaction always observes a consistent state of the system.

Design Choices

Although every TM system has to provide the guarantees enunciated before, 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

In order to provide the ACI guarantees mentioned before, 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 [28]. In environments where conflicts are frequent, this approach can be the best choice. Bartok STM [14], AUTOLOCKER [32] and McRT STM [38] are some examples of TM systems that use pessimistc concurrency control.

The other approach is optimistic concurrency control. When using optimistic concurrency control, 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. Techniques such as using a contention manager to delay the re-execution of transactions or aborting only transactions that conflict with other committed transactions may be used to avoid this problem. In environments 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. DSTM [19] and RSTM [30] use optimistic concurrent control.

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. Examples of TM systems that use eager version management are: Bartok STM [14] and McRT STM [38].

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. Examples of TM systems that use lazy version management are: TL2 [9], DSTM [19] and RSTM [30].

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.

Nesting

Some TM systems allow for a transaction to be defined inside another transaction. This is called a nested transaction. The semantics of nested transactions depend on which type of nesting is used in the TM system.

The simplest type of nesting is flat nesting. When using flattened nesting, commits made by the inner transaction will only be actually committed when the outer transaction commits. On the other hand, an abort caused by an inner transaction will also abort the outer transaction.

Another type of nesting is closed nesting. When using closed nesting, an inner transaction commit or abort will transfer the control to the outer transaction. An abort of the inner transaction will not cause the outer transaction to abort and a commit of the inner transaction will make the changes visible to the outer transaction only. The other running transactions will only see the changes when the outer transaction commits.

One more type of nesting is open nesting. When using open nesting, an inner transaction commit will become visible to the other running transactions, just like an outer transaction commit. Even when the outer transaction aborts, the results of the already committed inner transaction will remain committed.

Up until now, the nesting models we discussed only consider that one of the inner transactions of a transaction is running at the time. Parallel nesting changes this assumption by allowing multiple inner transactions of the same outer transaction to execute at the same time.

Software vs Hardware

Although TM was initially implemented in hardware, currently there are numerous software TM systems available. Even though software TM systems have an increased overhead, with recent TM systems, software TM is a reasonable choice. Despite the overhead of software TM, it has advantages over hardware TM. Software TM is more flexible and easier to modify, allowing for experimentation with new algorithms and extension of existent algorithms and can be easily integrated with existing systems and programming languages features [15].

Haswell Transactional Memory

In 2013, Intel released the Transactional Synchronization Extensions (TSX) extension to their x86 instruction set. TSX first appeared on selected CPU models of the Haswell microarchitecture [22, 23]. The TSX specification provides two different interfaces, one is Hardware Lock Elision (HLE) and the other is Restricted Transactional Memory (RTM).

HLE is based on previous work by Rajwar and Goodman [36] and it is backwards compatible with older CPUs that do not have TSX. HLE introduced two new instruction prefixes (XACQUIRE and XRELEASE). On CPUs without TSX, these prefixes are ignored and the instructions are executed normally. XACQUIRE is used in instructions that acquire a lock address and XRELEASE is used in instructions that release the lock address. After executing an instruction with the XACQUIRE prefix, the lock address is added to the read-set of a transaction but nothing is written to the address. This allows several concurrent threads to acquire the lock and execute the same critical section at the same time. The thread then runs the following instructions while keeping track of the read and written memory addresses and adding them to the read and write sets. If a conflict is detected, the CPU will discard any memory writes made prior to the XACQUIRE and it will re-execute the code since the XACQUIRE but it will actually write to the lock address. This write will also abort concurrent threads, since they read the lock address when starting. XRELEASE marks the end of the lock elision. After XRELEASE the CPU will try to commit the transaction, if it succeeds, the code between XACQUIRE and XRELEASE was executed without acquiring or releasing the lock.

RTM enables the programmer to use TM by adding three new instructions: XBEGIN, XEND and XABORT. The XBEGIN instructions starts a transaction, the XEND instruction ends a transaction and the XABORT instruction aborts a running transaction. Additionally, the XBEGIN instruction receives a fallback address that will be jumped to when an abort happens. The fallback address is needed because it is not possible for TSX to guarantee that the transaction will succeed, even in the absence of conflicts. This happens due to the limited nature of hardware TM [10, 26].

Despite the efforts made by Intel to introduce hardware TM in their CPUs, they faced a significant

step back. After the release of the CPUs that included TSX, a critical bug was discovered [24, 25]. This lead Intel to issue microcode updates to disable TSX on the affected CPUs.

Joining Thread-Level Speculation and Transactional Memory

Instead of choosing between a fully automatic approach and fine-grained parallelization, we argue that is is possible to get the benefits of both if we join TM and TLS in a single system. In order to use this system, the programmer would first hand-parallelize the program into several coarse-grained user-threads, using transactions to ensure atomicity where needed. Upon execution, the program would run with the TM semantics that the programmer expects, but will automatically split the user-threads into several tasks. These tasks would then run speculatively in parallel to further the parallelization. A task can be created either inside or outside of transactional code however, it cannot cross boundaries between transactional and non-transactional code. When a tasks execution causes a conflict with a previous task from the same user-thread or with a task from another user-thread, the task is aborted and its effects are not visible. Even though the tasks are allowed to run out-of-order, they still must behave as if they were running sequentially which means that they need to observe all the writes made by the previous tasks from the same user-thread when executing.

TLSTM

TLSTM [2] is based on SwissTM [11], which is a state-of-the-art software TM system that uses optimistic read-write conflict detection and pessimistic write-write conflict detection. Before explaining them TLSTM in more detail, we will start by explaining the basic concepts present in SwissTM. We will then explain the most important parts of TLSTM.

SwissTM

There is a commit counter, named commit-ts that is incremented by any transaction that writes, when it is committed.

Every memory location has two corresponding locks: a write lock (w-lock) and a read lock (r-lock). The w-lock holds either a write-log entry or the unlocked value. The r-lock either holds either a version number or the locked value.

Due to the pessimistic write-write conflict detection, when an user-transaction attempts to write to a memory location, it must first obtain the corresponding w-lock.

SwissTM uses lazy version management, which means that writes are first stored in the redo-log and are only written to memory when the user-transaction commits.

In order to prevent transactions from reading data that is still being committed and may be inconsistent, the r-locks of the locations written by an user-transaction are acquired during its commit. After a successful commit, the r-locks are unlocked and are updated with the new commit-ts value.

An user-transaction keeps a timestamp called valid-ts that denotes a point in the logical commit time for which all the values that the user-transaction has observed so far are guaranteed to be valid. When a value that has higher version than the user-transaction valid-ts is read, the valid-ts has to be revalidated, i.e., it needs to be updated to the version of the value being read. This validation consists in verifying that every entry in the read-log is still valid at the new valid-ts.

Conflict Detection

There are two types of conflicts that may happen between tasks: write-after-write (WAW) which happens when a task writes to a location written by a future task and read-after-write (RAW) which happens when a task writes to a location that a future task read.

In order to detect WAR conflicts, a new task validation procedure was added. This validation procedure starts by validating both SwissTM's redo-log and a new task-read-log that records the reads made by previous tasks. It will then check if any of the values read from committed state were speculatively written by a previous running task from the same transaction. After that it checks if the values the task has read from previous tasks have been updated by a future tasks from the same transaction. If either of these are true, the task performing the validation aborts. This validation may need to happen at read, write and commit time.

The detection of WAR conflicts also requires changes to the SwissTM's algorithm. If SwissTM's algorithm was used, intra-thread deadlocks could ensue when a task writes to a location and waits for the previous tasks to complete when trying to commit. If the previous task wants to write to the same location, both tasks would deadlock waiting for each other to release the w-lock. In order to solve this problem, SwissTM's write-write conflict handling was modified. Now, when a task wishes to acquire a write-lock that is held by a past task from the same user-thread, the task wishing to acquire the lock aborts. If, otherwise, it was a future task that write-locked the location, that future task will be signaled to abort.

When committing a task, TLSTM must also ensure that all previous tasks have completed and cannot be aborted because of intra-thread conflicts. This is achieved by serializing commits of tasks belonging to the same user-thread, along with the previously explained intra-thread conflict detection.

Transaction Commit

The transaction commit algorithm is very different from SwissTM's. this happens due to the fact that, in order to ensure atomicity, TLSTM needs to take into account the reads and writes performed by every single task of the user-transaction.

During commit, the reads of all the tasks in an user-transaction are validated by the last task in the transaction. This task then updates all the values written by the remaining tasks.

Transaction Abort

As with commit, the last task of the aborting user-transaction is also responsible for the abort. This task will clear every write-lock of all tasks in its user-transaction and reset the tasks' state to their last known correct values. Finally, the last task signals every past task of its user-transaction to restart, before restarting itself. When any other task receives the abort transaction signal it waits until all tasks from its user-transaction have received that signal.

Preventing Deadlocks

Suppose we have an application with two user-threads, each split into two tasks. The second task from the first user-thread has the w-lock for location X and the second task from the second user-thread has the w-lock for location Y. Now, if the first task from the first thread attempts to write to Y and the first task from the second thread attempts to write to X, both would have to wait for the other user-thread to commit or abort before getting the lock. However, the contention manager will not signal the lock owners to abort and they cannot commit either because they are waiting for their past tasks to complete (as a consequence of serializing commits). To solve this problem, the inter-thread contention manager must be modified to support tasks: when an inter-thread conflict is detected between two tasks, the contention manager should abort the more speculative one.

Inconsistent Reads

Due to running code concurrently, both TLS and TM create out of order reads. Suppose we have a user-transaction with two tasks. The first task writes NULL to location X before creating a new object and making X point to it. If the second task attempts to read X just after the first sets it to NULL, the program will crash. Although these problems do not occur in SwissTM due to the opacity criterion [12], when adding TLS, values can be read from running tasks, which may result in reading intermediate and incorrect values [33]. Therefore, in a unified runtime it is not possible to prevent all inconsistent reads, so TLSTM needs to detect and take care of those coming from TLS. This is done by checking if a location is valid before being read by a task. This verification is needed before every read, even for correct reads.

Chapter 3

Architecture

Although there is previous work on combining TM and TLS together (TLSTM [2]), our work has a very different approach to this problem. TLSTM is a new algorithm created by using an existing TM system and developing a new TLS around it. What we intend to do is combining different existing TM and TLS algorithms without changing either. This modularity will allow for someone who uses our system to use any TM or TLS systems like black boxes and to combine them for different settings without changing or understanding the existing algorithms. The main advantage of this modularity is that it would allow an average programmer to take advantage of new improvements made by TM and TLS research with very low effort.

The Problems of Joining TM and TLS

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.

In order to help us understand the problems we need to resolve when attempting to run TLS on top of TM, several examples of possible failures and our initial approach to resolve them are illustrated next.

One of the problems of performing shared reads and writes inside code parallelized by TLS is that the TM read and write functions where not meant to be called concurrently inside the same transaction. These functions might use thread local data that would need synchronization. Because we cannot modify the TM algorithm, calling these function would required the addition of locks before every shared read or write when inside TLS parallelized code. This would likely result in a massive performance decrease.

The second 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. Note that, the TM functions have to be bypassed for the whole transaction, not just for the TLS parallelized code. To understand why, consider the code in [Fig. 3.1]. In this case, if the TM uses lazy version management, the read of *x* made in line 4 will not read the value that was previously written in line 2.

```
1: atomic
2:     x ← 1                          ▷ Goes through TM write function
3:     for i ← 1, N do                ▷ Parallelized by TLS
4:         y ← y + x                  ▷ Reading of x goes through TLS read function
5:     end for
6: end atomic
```

Figure 3.1: Example of failure when mixing transactional and non-transactional memory accesses.

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 the following table:

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

Table 3.1: 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).

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.

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. 3.2]. 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 7. After line 7, 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:              for i ← 1, SIZE do                   ▷ Parallelized by TLS
6:                  localProcessedData[i] ← DoStuff(src[n][i])   ▷ DoStuff only performs reads
7:              end for
8:              SetInsideTLS(false)
9:              Copy(localProcessedData, dst[n], SIZE)
10:             SetAlreadyProcessed(n, true)
11:         end if
12:     end atomic
13: end function
```

Figure 3.2: 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.

We mentioned that it is not possible to write to shared variables before the TLS block. The reason for this is that, if the TM system uses lazy version management, we can get incorrect results. Suppose we write to a variable *x* before the TLS block. Then, inside the TLS block, we read the same variable. The value returned may not be the same that was written because it was written only on the TM redo-log. Since reads inside the TLS block do not use the TM functions, we will read an incorrect value. Even if the variable is not read inside the TLS block, it is still not possible to write to it before. To understand why, consider we have two shared variables: *x* and *y*. *x* is written to before the TLS block and *y* is read inside the TLS block. In our solution we keep track of the transactions performing writes and reads inside the TLS block to each shared variable. This is done to avoid having a TLS block read a variable being written by another transaction or vice-versa. Because it is not feasible to keep track of every possible memory location, sometimes two different variables will be identified as the same by our system. If this happens with *x* and *y*, after the write of *x*, our system will detect one transaction writing to both variables. Later on, when reading *y*, inside the TLS block, our system will see there is a transaction writing to it. Not knowing it is itself, it will deadlock waiting for the transaction to end.

We also mentioned that is not possible to perform a shared read on an address that is later on read inside the TLS block. To understand why, consider the the execution order in [Fig. 3.3]. If the TM system uses eager version management, this execution order would result in transaction 1 committing in spite of having read two different values for *x*. This happens because the reads inside TLS do not do through the TM_Read function.

In our simplified model, we choose not to allow shared writes inside TLS parallelized code. This might look like a big restriction at first but, in a lot of cases, it can be worked around easily. A shared variable write can often be transformed into a local variable write when run inside TLS parallelized code.

Transaction 1	Transaction 2
TM_Read(x) - returns 0	
	TM_Write(x, 42) COMMIT
TLS_Read(x) - returns 42 COMMIT	

Figure 3.3: Example execution order that causes wrong results when a shared read is performed before the TLS block and again inside the TLS block.

After the TLS execution ends, the used local variable can be written to the target shared variable using TM functions.

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 were 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 16 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 array position stores two values. The least significant 8 bits store the number of running TLS blocks that read the memory locations mapped to the specified array position. The most significant 8 bits store the number of running transactions that performed writes to the memory locations mapped to the specified array position.

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.

Additionally, the following transaction local variables are used:

- inside_tls - Boolean value that tells us if we are inside the TLS block.
- tm_written - Set of the addresses of shared variables written by the transaction.
- tls_read - Set of the addresses of shared variables read inside the TLS block.
- tr_lock - Lock used to allow concurrent access to the tls_read set.

All the local variables are inside a structure called tx_local.

The sets of read and written memory locations are needed because, when the transaction ends, we need to know which positions of the addr_counts array need to be updated. Additionally, it is also used to avoid writing to the addr_counts array more than once per read or written variable. The lock for the tls_read set is needed because this set may be written to concurrently due to being written inside TLS parallelized code.

Performing Shared Reads

The code of the TM_Read function can be seen in [Fig. 3.4]. 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 it is already present in our local set for variables read inside the TLS block (line 7). If it is not, we need to update the addr_counts position to signal other transactions that they cannot write to this address (lines 8-11). Additionally, we need to update the tls_read set with this address (line 12). This is needed because, at transaction end, we need to know which addresses we updated in addr_counts. If the address was already read inside this transactions TLS block, we know that addr_counts was already correctly updated and we can just return the read value (line 15).

```
1:  function TM_READ(addr)
2:      if not tx_local.inside_tls then
3:          return TM_Read_Original(addr)
4:      end if
5:      h ← Hash(addr)
6:      Lock(tx_local.tr_lock)
7:      if not Contains(tx_local.tls_read, h) then
8:          do
9:              counts ← addr_counts[h]
10:             writers_count ← Highest8Bits(counts)
11:         while writers_count > 0 or not CAS(addr_counts[h], counts, counts + ONE_TLS)
12:         Insert(tx_local.tls_read, h)
13:     end if
14:     Unlock(tx_local.tr_lock)
15:     return *addr
16: end function
```

Figure 3.4: Implementation of the TM_Read function.

It is important to ensure that the update of addr_counts (lines 8-11) happens before the update of local tls_read set (line 12). This is needed to ensure correctness because a TLS thread might update the tls_read set locally and then fail to update the addrs_counts array. In this situation, another TLS thread, of the same transaction, might see that the address is already in the local tls_read set and proceed to read the address without addr_counts having been updated. If this happens we could have another transaction performing a write on an address that is now being read inside the TLS block.

Also, notice that when performing a shared read inside the TLS block, our code reads the value of *addr* directly from memory instead of calling TM_Read_Original. This happens because of the impossibility of calling TM functions concurrently in TLS parallelized code explained in the previous section. Returning the value directly is still correct, because we ensure that no transaction is writing the value and any transaction that attempts to write to it will abort (see [Fig. 3.5], line 7). Furthermore, as we stated before, variables read inside the TLS cannot be read or written to before the TLS, so we know that this is the first time we are reading this address on this transaction.

Performing Shared Writes

The code of the TM_Write function can be seen in [Fig. 3.5]. 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 (line 3). If it did not, we can just call the TM write function (line 13). Otherwise, we need to update the addr_counts array (lines 4-10). The logic is similar to the case of a shared read inside the TLS block. Before updating addr_counts, we check if there are any transactions, other than this, performing shared reads in their TLS block (line 7). If there are, we abort the transaction (line 8), since we cannot write to variables being read by the TLS block of other transactions. Otherwise, addr_counts is updated (line 10) and we update the tm_write set with the address being written. Similarly to the read case,this set is used both at transaction end to update the addr_count array and on this function to check if we already wrote to the address.

```
1:  function TM_WRITE(addr, val)
2:      h ← Hash(addr)
3:      if not Contains(tx_local.tm_written, h) then
4:          do
5:              counts ← addr_counts[h]
6:              readers_count ← Lowest8Bits(counts)
7:              if readers_count > Count(tx_local.tls_read, h) then
8:                  TM_Abort_Original()
9:              end if
10:         while not CAS(addr_counts[h], counts, counts + ONE_WRITER)
11:         Insert(tx_local.tm_written, h)
12:     end if
13:     TM_Write_Original(addr, val)
14: end function
```

Figure 3.5: Implementation of the TM_Write function.

Finishing a Transaction

The code of the TM_End function can be seen in [Fig. 3.6]. 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 3-8). This is done by using the data in the the local transaction sets, tm_written and tls_read. Additionally, we also need to clear the values in these sets (lines 9-10).

```
1:  function TM_END
2:      TM_End_Original()
3:      for all h in tx_local.tm_written do
4:          addr_counts[h] ← addr_counts[h] − ONE_WRITER
5:      end for
6:      for all h in tx_local.tls_read do
7:          addr_counts[h] ← addr_counts[h] − ONE_TLS
8:      end for
9:      Clear(tx_local.tm_written)
10:     Clear(tx_local.tls_read)
11: end function
```

Figure 3.6: Implementation of the TM_End function.

It is important to note that the locations read inside the TLS block cannot be released until the transaction ends. Releasing them after the TLS block ends would give incorrect behavior. To understand why, consider the code in [Fig. 3.7]. If this order of execution happened and we released the lock right after the TLS block, the first thread would have committed in spite of reading two different values for the variable *x*. With variable *x* being released only after the transaction ends, the second transaction would abort when trying to write to *x*.

Transaction 1	Transaction 2
SetInsideTLS(true)	
TM_Read(x) - returns 0	
SetInsideTLS(false)	
	TM_Write(x, 42)
TM_Read(x) - returns 42	
Transaction commits.	Transaction commits.

Figure 3.7: Example execution order that causes wrong results when the TLS read lock is released before the transaction ends.

Lockless version

The previous implementation has a big performance bottleneck. While performing reads inside the TLS block, the accesses to the tls_read set need synchronization. We achieved it by using a lock. This is an easy approach that guarantees correctness. However, it is terrible for performance. Note that the synchronized section includes the loop used to ensure that we write the correct value to addr_counts [Fig. 3.4, lines 7-10]. This means that, if one of the TLS threads has to wait for another transaction to finish writing to a variable, the other TLS threads will wait for this one to finish. Even if they are reading a different variable that is not being written by any transaction. This hinders parallelism considerably.

In order to remove the need for this lock we made the following changes to the previous algorithm:

- The transaction local data (tx_local) now includes as many tls_read sets as the number of TLS threads and the tr_lock was removed. When TLS is being run, each set belongs to one of the TLS threads.

- The TM_Read function was changed to work with these changes [Fig. 3.8]. Now, it has no locks and uses only the tls_read set that corresponds to the TLS thread performing the read.

- The TM_End function was changed too [Fig. 3.9]. It will now read the values to update from all the tls_read sets and clear all sets.

Because each TLS thread has its own tls_read set, the lock is not needed anymore, since TLS threads no longer shared any transaction local data. This allows us to have concurrent TLS reads at a minor cost: previously, when several TLS threads read the same variable, only one CAS operation (to update addr_counts) needed to be performed. Now, each TLS threads that reads the same variable has to perform an update to addr_counts. Due to the increased parallelism, the benefits outweigh the costs.

```
1: function TM_READ(addr)
2:     if not tx_local.inside_tls then
3:         return TM_Read_Original(addr)
4:     end if
5:     h ← Hash(addr)
6:     if not Contains(tx_local.tls_read[tls_th], h) then
7:         do
8:             counts ← addr_counts[h]
9:             writers_count ← Highest8Bits(counts)
10:        while writers_count > 0 or not CAS(addr_counts[h], counts, counts + ONE_TLS)
11:        Insert(tx_local.tls_read[tls_th], h)
12:    end if
13:    return *addr
14: end function
```

Figure 3.8: Implementation of the TM_Read function in the lockless version.

```
1: function TM_END
2:     TM_End_Original()
3:     for all h in tx_local.tm_written do
4:         addr_counts[h] ← addr_counts[h] − ONE_WRITER
5:     end for
6:     for i ← 0, TLS_COUNT − 1 do
7:         for all h in tx_local.tls_read[i] do
8:             addr_counts[h] ← addr_counts[h] − ONE_TLS
9:         end for
10:    end for
11:    Clear(tx_local.tm_written)
12:    for i ← 0, TLS_COUNT − 1 do
13:        Clear(tx_local.tls_read[i])
14:    end for
15: end function
```

Figure 3.9: Implementation of the TM_End in the lockless version.

Improved Implementation

In this implementation we modified the addr_counts array to store, in each position, the id of all the threads performing writes or TLS reads to shared variables. In order to store this additional information, each position of addr_counts now takes 64 bits instead of 16. As in the previous implementation, 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. The difference is that, instead of storing just the count of transactions performing TM writes and TLS reads in each part, 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.

Since we can tell what locations were read and written by each transaction just by using addr_counts, the transaction local tm_written and tls_read sets can now be removed. This means that this implementation will only need the inside_tls variable as local transaction data.

The new implementation of the TM_Read function [Fig. 3.10] is very similar to the previous implementation. The differences are that instead of checking the local transaction sets to see if the read was already performed in by this transaction, we can get that information from the addr_counts array (lines

6-7). Additionally, the update of the transaction local read set is no longer present since this information is updated simultaneously with the addr_counts array (lines 8-11). Notice that with this solution we were able avoid the cost introduced by the lockless version. Now, when a TLS thread signals that it is reading a location, all the other TLS threads will see that the transaction is reading that location, because the same position is used for all TLS threads in the same transaction.

```
 1: function TM_READ(addr)
 2:     if not tx_local.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 3.10: Improved implementation of the TM_Read function.

TM_Write [Fig. 3.11] is also similar to the previous version. The differences are similar to the TM_Read function. The check to see if the transaction already wrote to the target address is now done with the addr_counts array (line 4). And there is no longer any transaction local set to be updated, since the update is done when addr_counts is updated (lines 6-14). In this version, both reads and writes are slightly faster, since not only we avoid writing to the transaction local sets but also the check to see if the transaction already wrote or read a location is a simpler operation, using only bit manipulation instructions.

```
 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 ≠ 0 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 3.11: Improved implementation of the TM_Write function.

The TM_End function [Fig. 3.12] is different from the previous versions. Since we do not have any transaction local data to tell us the locations that were read or written, we need to go through the whole

addr_counts array and zero the two bits in each position that belong to the current transaction (lines 6-8). 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. There is a trade-off between these two situations.

1: **function** TM_END
2: $TM\_End\_Original()$
3: $tm\_mask \leftarrow ShiftLeft(1, tx\_id + 32)$
4: $tls\_mask \leftarrow ShiftLeft(1, tx\_id)$
5: $both\_mask \leftarrow (tm\_mask \mid tls\_mask)$
6: **for** $i \leftarrow 0, ADDR\_COUNTS\_SIZE - 1$ **do**
7: $addr\_counts[i] \leftarrow addr\_counts[i]\ \&\ \sim both\_mask$
8: **end for**
9: **end function**

Figure 3.12: Improved implementation of the TM_End function.

One disadvantage of this algorithm is that it limits the number of concurrent threads (TLS and TM threads) to 64 in a 64-bit machine. Currently, this is not an issue as there are not many machines with more than 64 hardware threads available. However, in the future, this may limit the use of this algorithm.

Chapter 4

Implementation

In the previous chapter, we presented the general description of the algorithm to support running code parallelized by TLS inside a transaction. In this chapter we will discuss the specific implementation details. These details are related to some optimizations that can be made when the transaction ends. We also describe how we perform the updates to the addr_counts array when a transaction aborts. Additionally, we detail how we used the specific TM and TLS systems we choose to implement our algorithm.

Optimizing the Transaction End

In both the initial version and the lockless version of the algorithm, there is a possible optimization that can be done when a transaction ends. Suppose a transaction reads a variable inside the TLS block and, after the TLS block, writes to the same variable. The position for that variable in the addr_counts array will have one TM write and one (or possibly more, in the case of the lockless version) TLS reads. When a transaction ends, the TM_End function will update the positions of the array corresponding to the variables that the transaction read inside the TLS block or wrote. It does this by going through the local tm_write and tls_read sets, one a the time, and updating the positions it finds in the sets. When a transaction does a TLS read and a TM write to the same variable, the same position will be updated multiple times. Since the addr_counts array is shared by all threads, this update has to be an atomic operation. Because of that, unnecessary updates should be avoided. The code for the optimized version of TM_End can be seen in [Fig. 4.1] The optimization to avoid the unneeded updates consists of joining all the updates to be made on the addr_counts array in a local map (lines 3-10) before actually updating addr_counts (lines 11-13).

Detecting Transaction Aborts

After the end of a transaction, the TM_End function needs to be called regardless of the transaction having committed or aborted. However, when the TM systems decides to abort a transaction, it is not

```
1: function TM_END
2:     TM_End_Original()
3:     for all h in tx_local.tm_written do
4:         local_counts[h] ← local_counts[h] + ONE_WRITER
5:     end for
6:     for i ← 0, TLS_COUNT − 1 do
7:         for all h in tx_local.tls_read[i] do
8:             local_counts[h] ← local_counts[h] + ONE_TLS
9:         end for
10:    end for
11:    for all h in tx_local_counts do
12:        addr_counts[h] ← addr_counts[h] − local_counts[h]
13:    end for
14:    Clear(tx_local.tm_written)
15:    for i ← 0, TLS_COUNT − 1 do
16:        Clear(tx_local.tls_read[i])
17:    end for
18: end function
```

Figure 4.1: Implementation of the optimized TM_End in the lockless version.

possible to call TM_End after the abort. In order to solve this problem, we added a variable called previous_aborted to the transaction local data. When a transaction starts, it checks if the value of previous_aborted is true. If it is, calls TM_End. Then, the value of previous_aborted is set to true. After a transaction commits, the TM_End function will be called and it will set previous_aborted to false. This ensures that previous_aborted will be true when a transaction aborts. Also, notice that the case where the last transaction to run aborts and TM_End is not called cannot happen. This situation is impossible because a transaction that aborts will be retried until it commits (and calls TM_End after).

RSTM

In order to implement our algorithm, we chose to use RSTM [40] as the TM system. RSTM is an open source transactional memory library that allows the user to choose among several algorithms without needing to change their code. RSTM is easy to use for any programmer that is familiar with TM since the available function closely follow the traditional TM semantics.

To use RSTM, our application has to call the TM_SYS_INIT function at the start of the application and, when the application ends, the TM_SYS_SHUTDOWN function needs to be called. Additionally, each thread that uses transactions, needs to call the TM_THREAD_INIT at the start of the thread and TM_THREAD_SHUTDOWN when the thread ends. In order to start a transaction, we start a code block prefixed with TM_BEGIN(atomic). To end a transaction we close the block started after TM_BEGIN and add TM_END after closing it. During the execution of the transaction, shared reads and writes can be performed by using the TM_READ and TM_WRITE functions, respectively. Since malloc cannot be used inside a transaction without avoiding memory leaks, extra functions are provided in oder to dynamically allocate memory inside of transactional code. These functions are called TM_ALLOC and TM_FREE and provide the same semantics of malloc and free, respectively. In [Fig. 4.1] we can see an example of the changes needed for a function to use RSTM.

Listing 4.1: Example of function before and after modifications to use RSTM

```
void PushLeft(Queue *queue, void *value)
{
    Node *node = malloc(sizeof(*node));
    node->value = value;
    Node *leftSentinel = queue->left;
    Node *oldLeftNode = leftSentinel->right;
    queue->left = leftSentinel
    queue->right = oldLeftNode;
    leftSentinel->right = node;
    oldLeftNode->left = node;
}

void TM_PushLeft(Queue *queue, void *value)
{
    Node *node = malloc(sizeof(*node));
    node->value = value;
    TM_BEGIN(atomic) {
        Node *leftSentinel = TM_READ(&queue->left);
        Node *oldLeftNode = TM_READ(&leftSentinel->right);
        TM_WRITE(&queue->left, leftSentinel);
        TM_WRITE(&queue->right, oldLeftNode);
        TM_WRITE(&leftSentinel->right, node);
        TM_WRITE(&oldLeftNode->left, node);
    } TM_END
}
```

SpLip

In order to implement our algorithm, we chose to use SpLIP [33] as our TLS system. SpLip is a state of the art TLS system that was tested against several benchmarks. It was able to obtain an average of 74% of the speedup of the optimal hard parallelized versions of such benchmarks. In order to use SpLip, first we need to create the definition of the ThManager class, and choose the number of CPUs used and the number of iterations to unroll. In order to parallelize a loop, we need to define a new class that will inherit from the SpecMasterThread class. This class represents one of the threads that will run the loop speculatively and will have to implement the following methods:

- initVars - Used to initialize instance variables
- end_condition - Condition to tell if the loop has ended
- interaction_body - Code for the loop iteration that will run speculatively
- non_spec_iteration_body - Code for the loop iteration that will run sequentially

A sample of the code needed to define this class can be seen in [Fig. 4.2]

In order to read or write variables modified during the execution of the loop, the spedLD and specST methods of the parent class (SpecMasterThread) need to be used.

To initialize the speculation, each thread needs to be created using the allocateThread method, registered on the previously created ThManager instance and initialized. An example of this can be seen in [Fig. 4.3] After the initialization is done, the loop starts when the ThManager speculate method is called.

Listing 4.2: Class the represents the speculative threads in SpLip

```
class MySpecTh : SpecMasterThread<...>
{
    void initVars(int var1);
    int end_codition();
    void updateInductionVars();

    int iteration_body()
    {
        // Read a variable
        int foo = this->specLD<...>(bar)

        // Write to a variale
        this->specST<...>(bar, 42)
    }

    int non_spec_iteration_body();
}
```

Listing 4.3: Code to initialize the speculation of a loop in SpLip

```
for (int i=0; i < PROCS_NR; i++) {
        MySpecTh *thr = allocateThread<MySpecTh>(i);
        thManager.registerSpecThread(thr, i);
        thr->initVars(42);
}

thManager.speculate<MySpecTh>();
```

Chapter 5

Evaluation

In this chapter we will present the results obtained by our system. The main goal of this section is to answer the following questions:

- In which cases is it beneficial to add spec transactions to an existing TM application?
- In which conditions does each of the developed algorithms perform best?

In order to answer these questions, two different benchmarks were used to test our system. The first one is an already existent benchmark, present in the STAMP suite [3] 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 Benchmark 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 [Fig. 5.1]. 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 (lines 8-16). 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 (lines 19-30). 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% [33]), 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.

```
1: function ADDNEWRESERVATION
2:     numQuery ← RandomBetween(1, N)
3:     customerId ← RandomBetween(1, Q)
4:     for i ← 1, numQuery do                    ▷ Choose random types and Ids to reserve
5:         types[i] ← RandomChoice(CAR, FLIGTH, ROOM)
6:         ids[i] ← RandomBetween(1, Q)
7:     end for
8:     for i ← 1, numQuery do                    ▷ Find the most expensive item of each type
9:         if Exists(types[i], ids[i]) then
10:            price ← GetPrice(types[i], ids[i])
11:        end if
12:        if price > maxPrices[types[i]] then
13:            maxPrices[types[i]] ← price
14:            maxIds[types[i]] ← ids[i]
15:            isFound ← true
16:        end if
17:    end for
18:    ▷ Make the reservations
19:    if isFound then
20:        AddCustomer(customerId)
21:    end if
22:    if maxIds[CAR] then
23:        Reserve(CAR, maxIds[CAR])
24:    end if
25:    if maxIds[FLIGHT] then
26:        Reserve(FLIGHT, maxIds[FLIGHT])
27:    end if
28:    if maxIds[ROOM] then
29:        Reserve(ROOM, maxIds[ROOM])
30:    end if
31: end function
```

Figure 5.1: Code of the vacation operation to add new reservation.

Comparing the Different Algorithms

In this section, we focus on comparing the performance of the different algorithms among each other. In order to simplify the representation of these results, we chose to use 4 TM threads for all tests. After running the vacation benchmark with the needed changes to use our system, and with the different developed algorithms we measured the execution times seen in [Fig. 5.2].

Comparing the results of the several algorithms among each other gives us the expected results: the version with locks is the slowest one, by a large margin. The fastest version is the improved version and the second fastest is the lockless version.

Comparing Different Thread Combinations

In this section, we focus on studying the performance results of all algorithms 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, and with the different developed algorithms we measured the following execution times:

- Execution time with the improved algorithm in [Fig. 5.3]

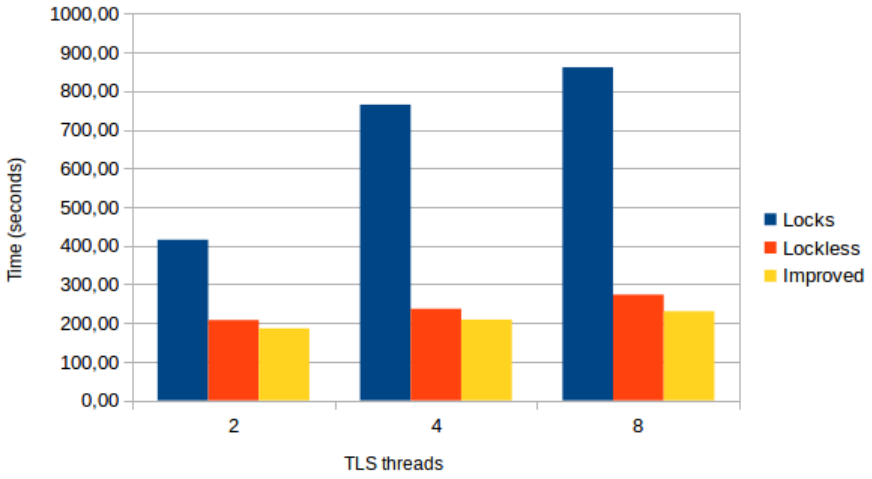

Figure 5.2: Execution times for vacation benchmark using 4 TM threads with 16 address bits.

- Execution time with the lockless algorithm in [Fig. 5.4]
- Execution time with the algorithm with locks in [Fig. 5.5]

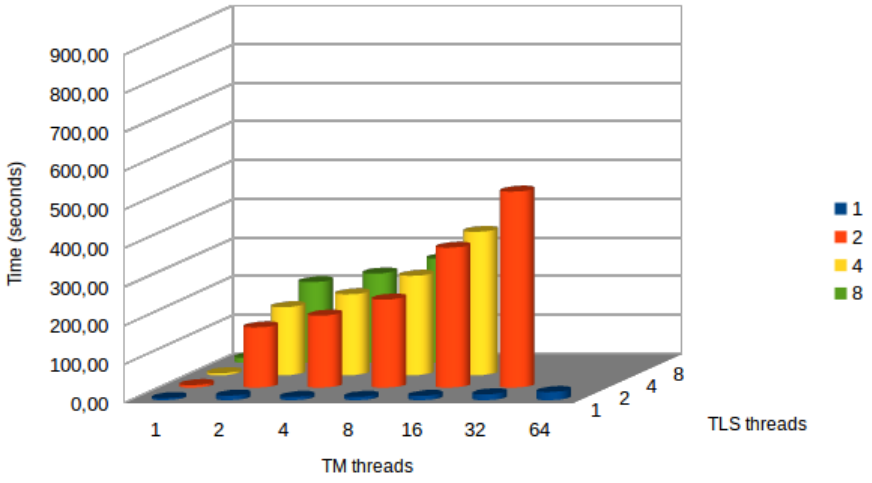

Figure 5.3: Execution time for vacation benchmark using the improved algorithm with 16 address bits.

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.

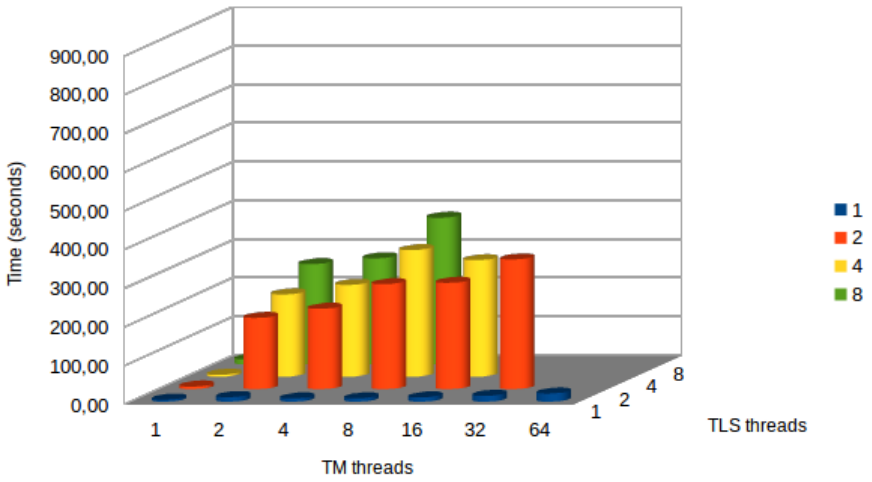

Figure 5.4: Execution time for vacation benchmark using the lockless version of algorithm with 16 address bits.

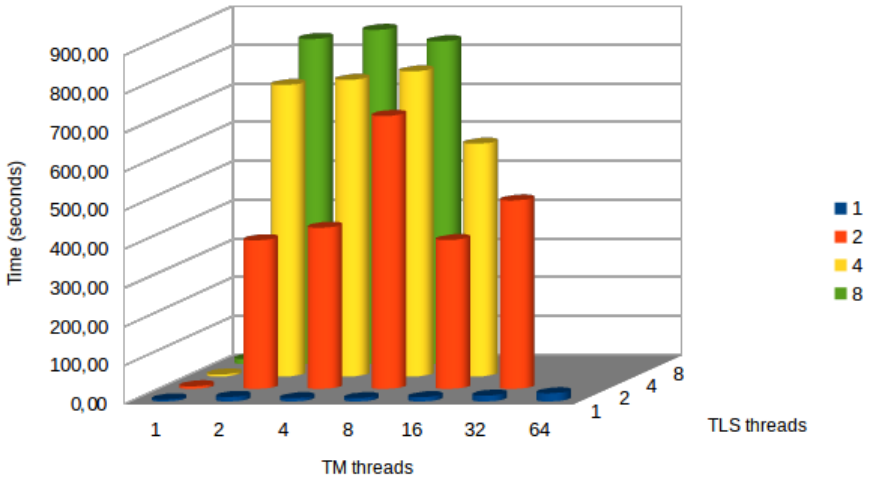

Figure 5.5: Execution time for vacation benchmark using the version of algorithm with locks with 16 address bits.

In addition to the execution time, we also measured the number of aborts caused by the introduction of TLS/TM conflicts:

- Aborts count with the improved algorithm in [Fig. 5.6]
- Aborts count with the lockless algorithm in [Fig. 5.7]
- Aborts count with the algorithm with locks in [Fig. 5.8]

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 parallelized 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

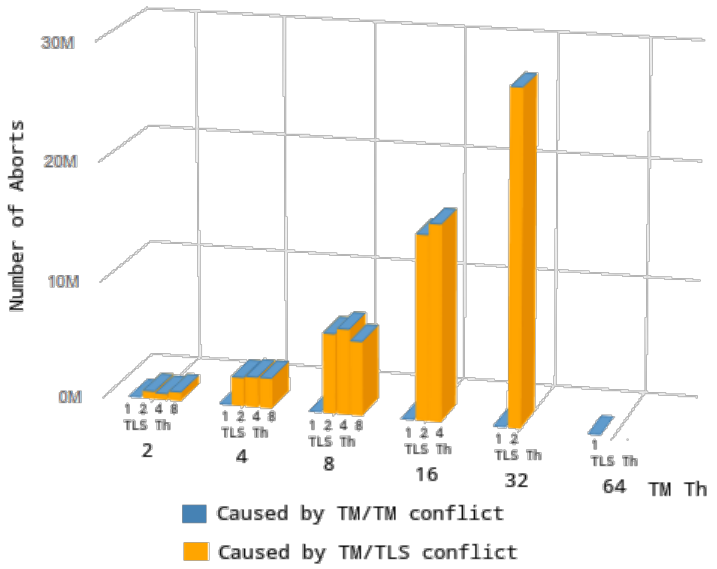

Figure 5.6: Transaction aborts for vacation benchmark using the improved algorithm with 16 address bits.

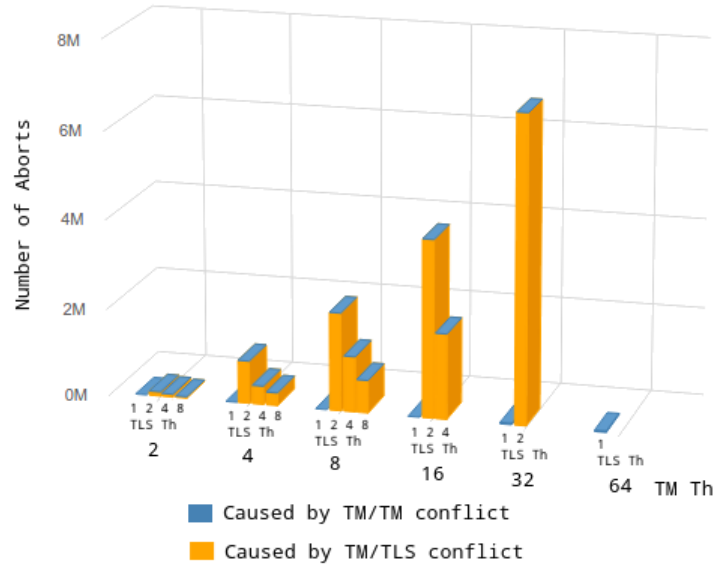

Figure 5.7: Transaction aborts for vacation benchmark using the lockless version of the algorithm with 16 address bits.

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 [29]. The data structures used by this

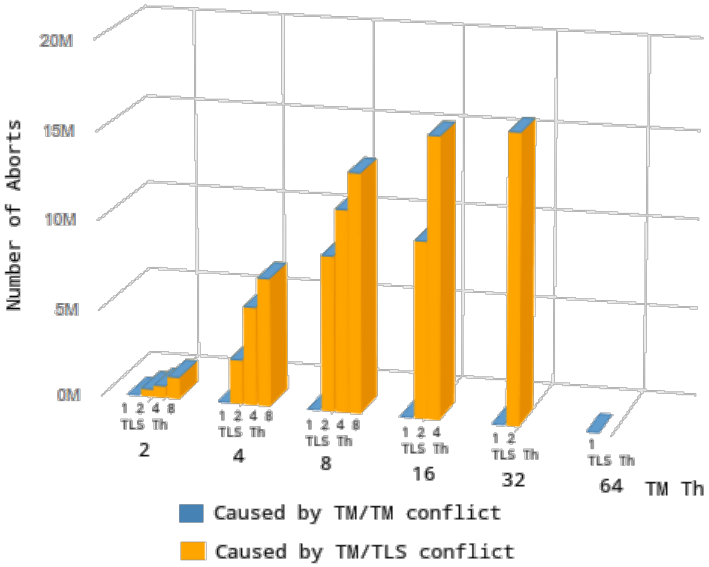

Figure 5.8: Transaction aborts for vacation benchmark using the version of the algorithm with locks with 16 address bits.

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 parallelized code 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

In this benchmark, the results obtained by the version with locks and the lockless version were identical, so we chose to only show the results for the lockless version. This happens because the difference between these two algorithms is in how they handle concurrent shared reads inside the TLS parallelized code. Since, as explained above, there are no shared reads inside TLS parallelized code, there is no difference between these two algorithms in this particular case. Additionally, the results showing the number of aborts caused by TLS parallelized code locking read variables are also not shown. These results were always 0 because, as explained above, the TLS parallelized code does not read any variable written by other transactions.

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

Comparing the Different Algorithms

In this section, we focus on comparing the performance of the different algorithms among each other. In order to simplify the representation of these results, we chose to use 4 TM threads for all tests. After running the red-black tree benchmark with the needed changes to use our system, and with the different developed algorithms we measured the execution times seen in [Fig. 5.9].

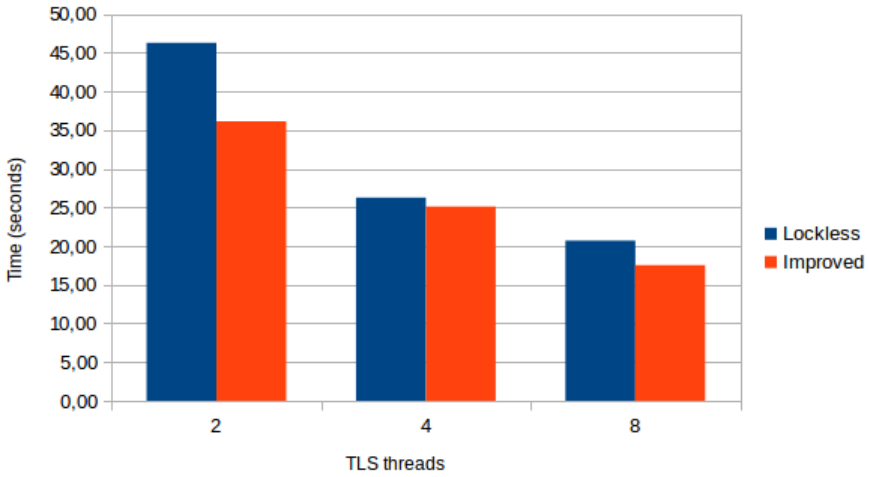

Figure 5.9: Execution times for red-black tree benchmark using 4 TM threads with 16 address bits.

Comparing the two different algorithms, we can see that improved algorithm is, as expected, a little more efficient than the lockless version. From these results we can conclude that, in the absence of shared reads inside the TLS parallelized code, adding TLS to an existing TM program can improve the performance.

Comparing Different Thread Combinations

In this section, we focus on studying the performance results of all algorithms 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, and with the different developed algorithms we measured the following execution times:

- Execution time with the improved algorithm in [Fig. 5.10]
- Execution time with the lockless algorithm in [Fig. 5.11]

In this benchmark, we were able to achieve positive results. With both algorithms, 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.

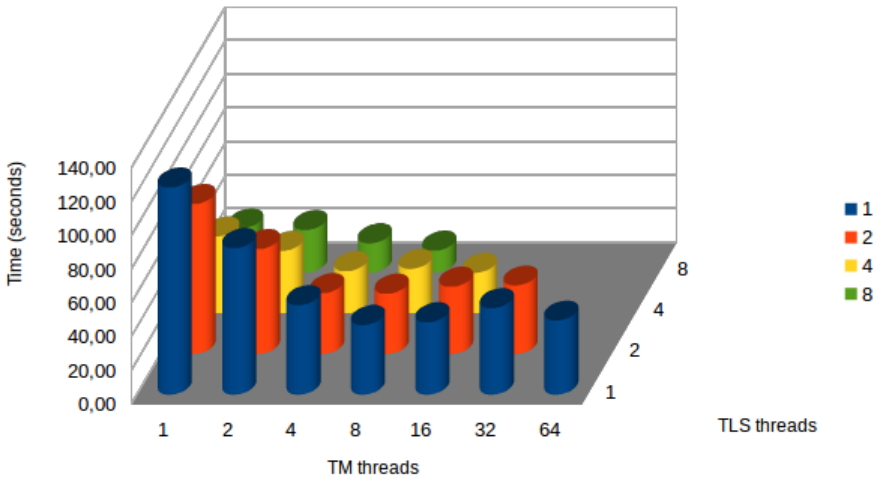

Figure 5.10: Execution time for red-black tree benchmark using the improved algorithm with 16 address bits.

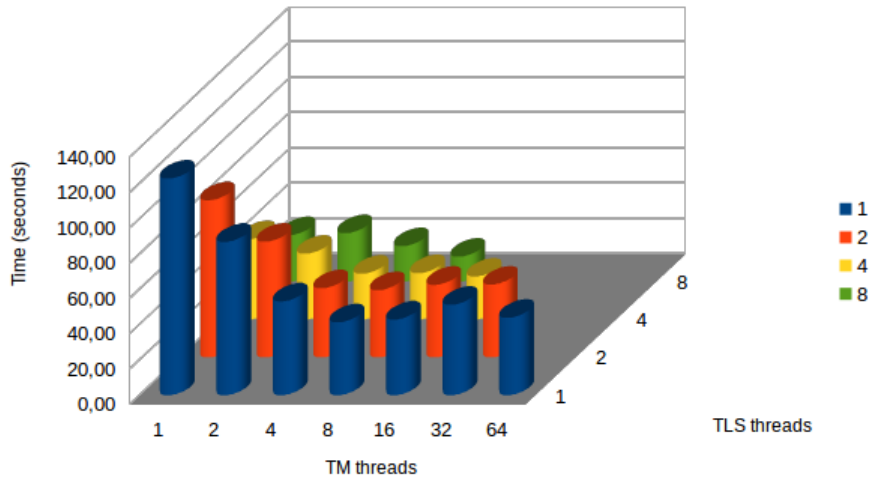

Figure 5.11: Execution time for red-black benchmark using the lockless version of the algorithm with 16 address bits.

Chapter 6

Conclusions

The widespread increase of the number of core in mainstream CPUs has lead increase the concerns about finding solutions to take advantage of this increase.

One of the solutions is TLS. TLS is an approach that is easy for a programmer to use, in some cases being fully automatic. However, the results obtained by TLS systems are only when positive when the number of dependences among TLS tasks is very low.

Another solution is TM. TM consists of having the programmer identify the sections of his code that can be run concurrently. While TM can achieve good results, it is not easy to make a TM program that can keep getting improved results with the increase of the available cores. Doing so, would require the programmer to split his program into very small transactions. This is difficult because it increases the number of possible execution orders exponentially. Since the programmer has to be sure that every possible execution order is correct, they will probably opt for the safe approach of using coarse-grained transactions, ensuring correctness while scarifying performance.

Instead of having the programmer choose between both approaches, we defend that should be possible to join TM and TLS together and obtain better results than either of the approaches alone could. Although there is previous work on joining TM and TLS together, our approach to this problem is substantially different. While previous systems try to add TLS to one existing TM algorithm, we allow the programmer to choose any TM and TLS algorithms he wishes.

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

6.1 Achievements

The main goal of this work was the development of a system that allows to programmer to join existing TM and TLS algorithms, without modifying either algorithm, with little effort.

The main achievements of this work are the following:

- *The study of different approaches to allow TM and TLS algorithms to be combined unmodified.* We developed two approaches to join TM and TLS. The correctness of these approaches and

their advantages and disadvantages were analyzed in detail, allowing for greater understanding of the difficulties faced when trying to join TM and TLS.

- *The development of a system that allows to join existing TM and TLS algorithms.* This is the main contribution of this work. After choosing the most promising approach, we developed a system that allows a programmer to join different TM and TLS algorithms with little effort. This allows programmer to easily take advantage of new TM and TLS research by replacing their TM and TLS algorithms with new and improved algorithms.

- *Analysis of the design decisions made.* While developing our system, the design decisions taken were studied, resulting in an detailed analysis of the advantages and disadvantages of each decision.

- *Benchmarking results.* To test our system, one existing benchmark was used an one new benchmark was developed. We were able to use both benchmarks to find cases where our system shows promising results and cases where further improvement could be made.

6.2 Future Work

In this section we discuss some possible future developments that could be made to improve our work. Some of the improvements are related shortcomings found and some are new possible research directions that were never related to the goals of our work.

One shortcoming of our work is that the benchmarks used test two extreme cases: the vacation benchmark tests the case where the TLS parallelized code reads most of the data shared among transactions and the red-black tree benchmark tests the case where the TLS parallelized code does not read any data shared among transactions. There is room for a new benchmark that theres the middle-ground between these two extremes. A new benchmark that would allow to configure the percentage of shared data that the TLS parallelized code reads. This benchmark could be used to determine the point where the shared date between the TLS parallelized code and other transactions becomes too much to be able to achieve positive results.

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 are useful is unrelated to our work. However, it might be worthy of more research effort.

Bibliography

[1] K. Agrawal, J. T. Fineman, and J. Sukha. Nested parallelism in transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 163, Salt Lake City, Utah, USA, February 2008.

[2] 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. ISBN 978-3-642-35169-3. URL `http://dl.acm.org/citation.cfm?id=2442626.2442639`.

[3] 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.

[4] 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. ISBN 978-1-60558-121-7. doi: 10.1145/1435523.1435526. URL `http://doi.acm.org/10.1145/1435523.1435526`.

[5] J. chun Wang, V. Adve, V. S. Adve, J. Mellor-crummey, J. Mellor-crummey, M. Anderson, M. Anderson, K. Kennedy, K. Kennedy, J. chun Wang, D. A. Reed, and D. A. Reed. An integrated compilation and performance analysis environment for data parallel programs. In *in Proceedings of Supercomputing '95*, pages 1370–1404, 1995.

[6] L. Codrescu and D. S. Wills. On dynamic speculative thread partitioning and the mem-slicing algorithm. In *IEEE PACT*, pages 40–46. IEEE Computer Society, 1999. ISBN 0-7695-0425-6. URL `http://dblp.uni-trier.de/db/conf/IEEEpact/IEEEpact1999.html#CodrescuW99`.

[7] M. Cornero. Mpsoc's in mobile phones: parallel computing for the masses. In *10th International Forum on Embedded MPSoC and Multicore*, 2010.

[8] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *ICPP*, pages 836–844, 1986. URL `http://dblp.uni-trier.de/db/conf/icpp/icpp1986.html#Cytron86`.

[9] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006.

Springer-Verlag. ISBN 3-540-44624-9, 978-3-540-44624-8. doi: 10.1007/11864219_14. URL http://dx.doi.org/10.1007/11864219_14.

[10] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 157–168, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-406-5. doi: 10.1145/1508244.1508263. URL http://doi.acm.org/10.1145/1508244.1508263.

[11] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 155–165, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542494. URL http://doi.acm.org/10.1145/1542476.1542494.

[12] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: 10.1145/1345206.1345233. URL http://doi.acm.org/10.1145/1345206.1345233.

[13] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. *SIGPLAN Not.*, 33(11):58–69, Oct. 1998. ISSN 0362-1340. doi: 10.1145/291006.291020. URL http://doi.acm.org/10.1145/291006.291020.

[14] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. *SIGPLAN Not.*, 41:14–25, June 2006. ISSN 0362-1340. doi: 10.1145/1133255.1133984. URL http://doi.acm.org/10.1145/1133255.1133984.

[15] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010. ISBN 1608452352, 9781608452354.

[16] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2002. ISBN 1-55860-596-7.

[17] 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. ISBN 0-8186-3810-9. doi: 10.1145/165123.165164. URL http://doi.acm.org/10.1145/165123.165164.

[18] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 0123705916, 9780123705914.

[19] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM. ISBN 1-58113-708-7. doi: 10.1145/872035.872048. URL http://doi.acm.org/10.1145/872035.872048.

[20] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. ISSN 0164-0925. doi: 10.1145/78969.78972. URL http://doi.acm.org/10.1145/78969.78972.

[21] J. Howard, S. Dighe, Y. Hoskote, S. R. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. F. V. der Wijngaart, and T. G. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *ISSCC*, pages 108–109. IEEE, 2010. ISBN 978-1-4244-6033-5. URL http://dblp.uni-trier.de/db/conf/isscc/isscc2010.html#HowardDHVFRJWBSPJJYMSEKRDLGAHLSBDWM10.

[22] Intel. Intel architecture instruction set extensions programming reference. Technical Report 319433-014, Intel, August 2012. URL http://download-software.intel.com/sites/default/files/319433-014.pdf.

[23] Intel. Transactional synchronization in haswell, 2012. URL https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell. [Online; accessed 11-November-2014].

[24] Intel. Desktop 4th generation intel core processor family, desktop intel pentium processor family, and desktop intel celeron processor family - specification update, 2014. URL http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/4th-gen-core-family-desktop-specification-update.pdf. [Online; accessed 11-November-2014].

[25] Intel. Intel xeon processor e3-1200 v3 product family - specification update, 2014. URL http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf. [Online; accessed 11-November-2014].

[26] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for ibm system z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 25–36, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4924-8. doi: 10.1109/MICRO.2012.12. URL http://dx.doi.org/10.1109/MICRO.2012.12.

[27] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. ISBN 1-55860-286-0.

[28] 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. ISBN 978-1-59593-973-9. URL http://dblp.uni-trier.de/db/conf/spaa/spaa2008.html#KoskinenH08a.

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

[30] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of software transactional memory. In *ACM SIGPLAN Workshop on Transactional Computing*. Jun 2006. Held in conjunction with PLDI 2006. Expanded version available as TR 893, Department of Computer Science, University of Rochester, March 2006.

[31] P. Marcuello and A. González. Thread-spawning schemes for speculative multithreading. In *HPCA*, pages 55–64. IEEE Computer Society, 2002. ISBN 0-7695-1525-8. URL http://dblp.uni-trier.de/db/conf/hpca/hpca2002.html#MarcuelloG02.

[32] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: Synchronization inference for atomic sections. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 346–358, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2. doi: 10.1145/1111037.1111068. URL http://doi.acm.org/10.1145/1111037.1111068.

[33] 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. ISBN 978-1-60558-606-9. doi: 10.1145/1583991.1584050. URL http://doi.acm.org/10.1145/1583991.1584050.

[34] 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. ISBN 0-7695-0425-6. URL http://dblp.uni-trier.de/db/conf/IEEEpact/IEEEpact1999.html#OplingerHL99.

[35] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, Oct. 1979. ISSN 0004-5411. doi: 10.1145/322154.322158. URL http://doi.acm.org/10.1145/322154.322158.

[36] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1369-7. URL http://dl.acm.org/citation.cfm?id=563998.564036.

[37] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 114–123, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-978-4. doi: 10.1145/1356058.1356074. URL http://doi.acm.org/10.1145/1356058.1356074.

[38] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the*

Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '06, pages 187–197, New York, NY, USA, 2006. ACM. ISBN 1-59593-189-9. doi: 10.1145/1122971.1123001. URL http://doi.acm.org/10.1145/1122971.1123001.

[39] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *25 Years ISCA: Retrospectives and Reprints*, pages 521–532, 1998. URL citeseer.nj.nec.com/sohi95multiscalar.html.

[40] J. Sreeram, R. Cledat, T. Kumar, and S. Pande. Rstm: A relaxed consistency software transactional memory for multicores. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 428–, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2944-5. doi: 10.1109/PACT.2007.62. URL http://dx.doi.org/10.1109/PACT.2007.62.

[41] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005. URL http://www.gotw.ca/publications/concurrency-ddj.htm.

[42] R. M. Yoo and H. S. Lee. Helper transactions: Enabling threadlevel speculation via a transactional memory system.

[43] 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. ISBN 978-1-60558-397-6. doi: 10.1145/1504176.1504183. URL http://doi.acm.org/10.1145/1504176.1504183.