TEL-AVIV UNIVERSITY
RAYMOND AND BEVERLY SACKLER
FACULTY OF EXACT SCIENCES
SCHOOL OF COMPUTER SCIENCE

Coping With Context Switches in Lock-Based Software Transacional Memory Algorithms

Dissertation submitted in partial fulfillment of the requirements for the M.Sc. degree in the School of Computer Science, Tel-Aviv University

by

Yoav Cohen

The research work for this thesis has been carried out at Tel-Aviv University under the supervision of Prof. Yehuda Afek

July 2011

Abstract

This thesis contributes to the investigation of lock-based software transactional memory (STM) algorithms, in multi-core shared memory systems. Lock-based software transactional memory algorithms do not perform well in workloads with a high rate of context switches, which is caused for example by scheduling events or page faults. This occurs since threads that are switched-out by the operating system while holding locks block other threads from progressing, causing their transactions to abort repeatedly.

We present here *Lock Stealing*, a novel contention management algorithm for minimizing the effect of context switches by enabling threads to acquire locks which are held by other threads. While some methods addressing this problem exist (e.g., `schedctl` in Solaris) they are *best effort* and only cover scheduling related context switches. In addition, they are platform specific and thus are not suitable or available in managed runtimes such as Java or .NET. In contrast, our approach is solely based on user-level code and is de-coupled from specific operating system events. We evaluate the performance of our approach on a set of benchmarks and observe improvements in both micro benchmarks and more elaborate test applications.

Contents

Chapter 1

Introduction

1.1 Background

The transactional memory (TM) [12] programming model promises to simplify concurrent programming by having the programmer specify *atomic* blocks of code (transactions), with the TM subsystem ensuring that these blocks run safely in parallel without deadlock. The performance of software TM implementations (STMs [21]) has received considerable attention by the research community, because STMs are unlikely to be adopted in practice if they add prohibitive overheads to applications. This has resulted in a shift from obstruction-free [10] STMs (e.g., DSTM [11]) to lock-based *blocking* STMs [4, 3, 7]. Due to the even weaker progress guarantee, lock-based STMs impose less book-keeping, data duplications and memory accesses on the application, and therefore show superior performance and scalability compared to obstruction-free STMs [3, 6].

Despite their superior performance in most cases, there are some situations in which lock-based STM algorithms are susceptible to blocking. In a lock-based STM, each section of memory [1] is associated with a lock. Transactions coordinate their accesses to shared memory by acquiring the locks that are associated with the sections in memory they access. Therefore, a transaction that needs to access a locked section of memory must either stall or abort. In an application with a high-degree of parallelism, such conflicts occur rarely and therefore do not cause performance issues. However, factors external to the STM can cause lock hold times to increase dramatically, leading to extremely poor performance. Here we are concerned with blocking or aborts that occur as a result of

[1] This can be a consecutive block of memory in *word-based* STMs or an object in *object-based* STMs.

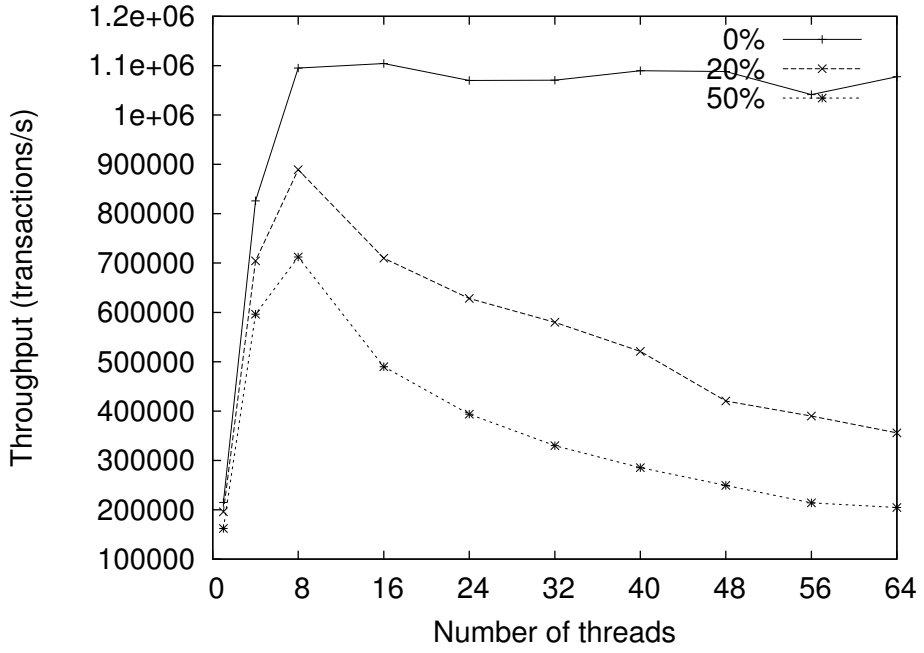

Figure 1.1: Throughput of a 64K sized red-black tree integer set with 0%, 20% and 50% update operations, on an Intel Nehalem with 8 concurrent hardware threads.

context switching a thread that is holding locks. The effect of this phenomenon depends on system specific properties such as the length of the scheduler's time slice, processor clock speed, processor cache configuration and others. For scheduling induced context switches the problem is significantly severe as a single context switch can potentially lead to the abort of hundreds of transactions, since the scheduler's time slice is usually in the scale of dozens of milliseconds and the execution of an average transaction in a realistic application is in the scale of dozens of microseconds.

To validate our hypothesis that repeated aborts cause throughput drop, we tested the correlation between the contention level of the application and the degradation of performance. We measured the throughput of a red black tree integer set benchmark using the TL2 lock-based STM [3] which is delivered as part of the Deuce STM platform for Java [13] on an Intel Nehalem with 8 concurrent hardware threads [2]. As our base-line, we used a benchmark with no contention at all (all operations are read-only and thus no locks are taken) and two benchmarks with 20% and 50% update operations, representing a medium contention workload and a high contention workload respectively. In order to control the rate of context switches, we measured each benchmark with a varying amount of threads, from 1 to 64, thus when running more than 8 threads context switches

[2]For details regarding our test environment see Section 4.

become frequent. Figure 1.1 shows the effect of scheduling induced context switches with relation to the contention level of the benchmark. In the base-line benchmark, as no locks are taken, there are no lock-holding threads that are switched-out by the OS and no performance degradation is observed. However, in the other benchmarks when running more than 8 threads, we observe that throughput degrades considerably.

While it would seem that it is possible to minimize the frequency of context switches to a certain extent by fixing the number of threads to equal the number of CPU cores (indeed, this is how STMs are often evaluated in the literature), we argue that this is not a viable approach in practice. First, the number of threads in the application is often dictated by various factors (e.g., servers with thread-per-request model) and bounding it places an undue burden on programmers. Even for applications where the number of threads can be limited, additional applications might be running on the same system, in different processes or even in the same process, putting pressure on the scheduler. The latter is in fact the case in modern application servers such as JavaEE servers. Finally, the managed code environments we are concerned with have their own set of threads to handle tasks such as garbage collection and just-in-time compilation.

Our main contribution in this thesis is therefore *Lock Stealing*, a purely algorithmic technique, requiring no special OS support, that allows transactions in lock-based STMs to make progress even when encountering a held lock, thereby mitigating the problem of threads that are switched-out by the OS while holding some of the STM's locks.

Lock stealing makes contention management [19] more applicable to lock-based STMs. Contention managers (CM) were introduced as a means to ensure progress of the STM system. A contention management algorithm decides which transaction to abort in case two transactions attempt to access the same block of memory. It does so by instructing the STM system to do one of the following:

- RESTART - instructs the STM system to abort the current transaction and try to execute it again.
- RETRY - instructs the STM system to try to access the memory again.
- ABORT OTHER - instructs the STM system to abort the other transaction and try to access

the memory again.

Current contention management approaches are most effective when used in obstruction-free STM algorithms. In an obstruction-free STM algorithm a transaction does not have to wait for another transaction to release a resource before it can acquire it. This important property enables the use of the ABORT OTHER action. However, in lock-based STM algorithms this is not the case – a transaction cannot continue if it needs to acquire an already held lock. At best a transaction can abort the other transaction but wait for it to release its locks. Thus a different approach is required. In Lock Stealing, if a transaction aborts another transaction it will simply take ownership of the lock without coordinating this with the other transaction. The other transaction will rollback gracefully and will not interfere with the first transaction.

1.2 Related work

Researchers have proposed ideas similar to lock stealing to simplify nonblocking programming [9] and to facilitate using efficient lock-based STM techniques in nonblocking STMs [15]. These works improve existing capabilities in nonblocking STMs while we introduce new capabilities to lock based STM (i.e., aborting other transactions and progressing) which have not yet been introduced in lock-based STMs.

The *revocable locks* of Harris and Fraser [9] are used to pass ownership of a transaction descriptor from thread T_2 to T_1 so that T_1 can *help* T_2 commit if they conflict. In contrast, we show how to pass ownership of a blocking lock from one thread to another for the purpose of aborting the original owner and progressing. Additionally, since we are focused on lock-based STMs our technique is more efficient than techniques in the nonblocking setting. Both [9, 15] require *indirection*: values of memory locations may reside in some private buffer and not in the actual location. In [9], the data at the location is actually modified to indicate indirection is required, whereas in [15] a separate orec table is used. Indirection increases the cost of memory accesses; complicates privatization; and implies that non-transactional code is either not allowed to access data modified by transactions, or that the non-transactional code needs to be instrumented to follow the STM protocol. Lock stealing has none of these limitations. Additionally, unlike [9] which performs revocations using OS

primitives, lock stealing uses only CAS operations. This makes it relatively fast and applicable in any OS and in managed runtimes like Java.

RobuSTM [23] is a lock-based STM that includes the ability to steal the locks of a running transaction. The motivation for stealing in RobuSTM is *robustness*, i.e., ensuring that *some* transaction always commits. Thus, RobuSTM only allows one thread at a time to steal a lock. Our motivation — throughput in the face of preemption — differs from [23]. Thus we allow for any number of threads to steal locks concurrently. In addition, RobuSTM relies on an x86-only atomic OR to guarantee stealing success, which isn't suitable for managed runtimes that cannot rely on hardware features which are not available in all common platforms.

Another related line of work is that of using *preemption deferral* mechanisms to prevent lock-holding threads from being switched out by the operating system [3, 14]. The idea is that when a thread acquires a lock it notifies the operating system's scheduler via a thread-local data-structure that it wishes to avoid being switched out. When the thread commits the transaction it returns to its normal state. TL2 uses the `schedctl` system API of Solaris to implement this functionality [3] while [14] implemented an extension to the operating system's scheduler in Linux, as Linux does not have a `schedctl`-like mechanism. Both approaches yield good results yet suffer from a few problems that lock stealing addresses.

To avoid misuse, preemption deferral mechanisms are *best effort* and the scheduler is free to ignore a request to defer preemption. If this happens, transactions are forced to *serialize* behind the switched out lock holder and the STM's performance will suffer. In addition, redistributing the time-slices between threads might lead to fairness issues with other processes that are running on the same system, in data-center applications for example. Furthermore, preemption deferral only handles context switches due to the fact that the thread used up its time-slice. If a thread is resolving a page fault while holding locks the context switch is unavoidable as the thread cannot continue executing before the page fault is resolved, and the system would again block. Similarly, if an interrupt is raised from the hardware while a thread is holding a lock, the user-level thread will stall while the operating system handles the interrupt, so preemption deferral provides no relief for this scenario as well.

Finally, preemption deferral is a platform specific mechanism. Providing this mechanism to platform agnostic frameworks such as Java is not trivial; it affects the application's portability

to other platforms and even if provided, will increase the complexity of the application. For applications with little portability requirements such as embedded devices this is not a problem, but for other applications, such as games, business and desktop applications portability and simplicity are major requirements.

Lock stealing is an opposite heuristic to preemption deferral: a transaction that conflicts with another transaction prefers to abort the other transaction instead of waiting for the other transaction to commit. Lock stealing is a completely algorithmic user-level technique and requires no special OS support, as evidenced by our Java implementation of it. Finally, because lock stealing hands the responsibility of making progress back to the transaction that encounters a held lock, it can also handle context switches due to page faults or interrupts.

An alternative approach to mitigate the problem of repeated aborts is yielding when faced with a switched-out thread that holds a lock. A thread may attempt to acquire a lock several times and if fails, can yield its time slice back to the scheduler. However, while reducing the number of aborts this approach also impacts throughput by reducing the number of transactions that are started. In addition, there is no guarantee that the next thread to be scheduled by the scheduler would not encounter the same conflict. Again, lock stealing is an opposite heuristic as it instructs threads that are running to make progress when faced with contention rather than give up their time slice.

Steal-on-abort is another notion of "stealing" (distinct from our work) that was proposed by Ansari et al. [1]. They suggest aborted transactions should be stolen by their opponents and serialized after them to prevent repeated conflicts. But in our oversubscription settings, the opponents are likely preempted and cannot do this stealing. Steal-on-abort has additional limitations that we do not suffer from: (1) it requires a model where threads submit their transactions to a thread pool for execution, and (2) it requires visible accesses to detect conflicts between two active transactions.

Chapter 2

Lock Stealing

The goal of Lock Stealing is to enable a running transaction to abort other transactions that hold locks it requires and acquire these locks without having to wait for the other transactions to release them. In this section we will present the Lock Stealing algorithm which is an enhancement that can be applied to most lock-based STM algorithms we are aware of. We will begin with a description of a general lock-based STM algorithm.

2.1 Lock-based STMs

A lock-based STM algorithm associates a lock with a section of memory. In *word-based* STMs, each lock is associated with a subset of the addresses in the application's address space; in *object-based* STMs, each object has its own lock. These locks are *versioned*: when unlocked, the lock's value represents a logical time when the associated memory was last written[1]. When locked, the lock points to the owner transaction (this is usually implemented by reserving the lock's least significant bit to specify the lock's state.)

Transactions execute transactional code (a code is usually defined to be transactional by the programmer) while intercepting accesses to memory. Transactional reads check the associated lock before proceeding: if it is locked, the transaction must stall or abort. Otherwise, the transaction reads the value from memory and rereads the lock to ensure its version has not changed, thus guaranteeing the value it observed is consistent. In addition, many STMs (e.g., [3, 7, 5]) validate

[1]Logical time can be maintained using a global counter [3, 7] or other mechanisms [17].

that the value read is consistent with some snapshot of memory. If it is not, the transaction aborts. (This, however, is not a fundamental property for supporting Lock Stealing and we describe it only for completeness.)

For transactional writes, two approaches exist. In a *lazy acquisition* STM, writes to memory are buffered in a write-set. In the transaction's commit protocol (explained shortly), the locks associated with each word written are acquired, the transaction copies the data from its write-set to memory, and then releases the locks. In an *eager acquisition* STM, locks are taken and the data is written to memory while the transaction is executing. This causes locks to be held for a longer period of time, but lets the transaction read its own writes without looking them up in a write-set (on abort, the transaction must *roll back* its updates and release the locks). Following [22], which has shown that write-sets can be implemented efficiently and with negligible performance loss compared to eager STMs, we assume a lazy acquisition STM from here on. Note that hybrid variations of these two approaches exist, such as eager acquisition of locks to facilitate early conflict detection, while still buffering writes in a write-set and deferring actual updating of memory to commit time [5]. For the purpose of our exposition, however, the crucial question is whether writes update memory only in commit time (lazy) or while the transaction is active (eager), so we consider such a hybrid approach as lazy STM.

Once the transaction completes the execution of the transactional code it begins a *commit protocol* in which it verifies that all the values it read from memory still form a valid atomic snapshot at the current logical time and if so, it commits by copying its write-set to memory and releasing the locks it held by updating their version number to the current logical time. Otherwise, the transaction is aborted and started again.

2.2 Transaction Life Cycle

We define three states of a transaction that we later use to determine whether the transaction is eligible for lock stealing:

- RUNNING - the transaction is executing the transactional code. Once finished executing the transactional code the transaction attempts to commit. In this state the transaction may be aborted by another transaction.

- COMMITTED - the transaction finished executing the transactional code and has either updated all the relevant memory addresses with new values or is about to do so. In this state the transaction may not be aborted and none of its locks may be stolen.

- ABORTED - the transaction is no longer running and is in the process of rolling itself back. It may have released some or all of its locks. In this state, other transactions may steal one or more of its locks.

We consider the time frame from the point a transaction starts to the point when a transaction is COMMITTED as the "Window of Abortability" and the time frame from the point when a transaction is COMMITTED to the point where it released all of its locks as the "Window of Un-Abortability". A transaction that encounters a conflict with another transaction that is in its "Window of Abortability" may resolve this conflict using Lock Stealing. However, if the other transaction is in its "Window of Un-Abortability" there is no option to recover from the conflict other than restarting itself and losing all the work it had done so far or risk waiting for a long period of time.

2.3 The Lock Stealing Algorithm

To facilitate the lock stealing algorithm we enhance certain data structures that are managed by the STM. We enhance each thread with a thread-local variable named `statusRecord` that combines the state of the transaction (RUNNING, COMMITTED or ABORTED) and a clock value of the thread. The clock value of the thread is initialized to 0 and incremented whenever a transaction is started. The goal of the `statusRecord` variable is to enable atomic updates of both the state of the transaction and its local clock value, using a `compare-and-swap` (CAS) operation. A discussion on the importance of the local clock value is given in Section 2.4. We enhance each lock variable to contain the identity of the lock owner and the owner's clock value. In some STM algorithms the lock variable may contain additional information, such as the version number of the lock [3, 7].

We begin by describing the lock acquisition protocol of the lock stealing algorithm, following the pseudo-code in Algorithm 1. When a transaction accesses a memory address it attempts to

acquire the lock that is associated with that memory address. If the lock is held by another transaction, the transaction consults its contention manager to decide how to proceed. If the other transaction is in its "Window of Un-Abortability" the contention manager may only decide whether to RETRY or RESTART. Otherwise, it may decide to instruct the transaction to take the ABORT OTHER action. If so, the transaction tries to abort the other transaction by atomically replacing its `statusRecord` variable to ABORTED using a CAS. If the CAS is successful, the transaction may attempt to steal the lock from the other transaction. See Algorithm 3 for a description of a simple contention manager. More elaborate contention managers will be discussed in Section 3.

To steal the lock the transaction tries to CAS the lock variable to a new value, that contains its thread id and clock value. If the CAS is successful the transaction may continue executing the transactional code. If not, the transaction is restarted (the transaction may decide to try to steal the lock again at this point instead of restarting immediately).

We continue with the transaction's commit protocol following the pseudo-code in Algorithm 2. If a transaction decides it can commit according to the lock-based STM's commit protocol, it attempts to atomically set its `statusRecord` to COMMITTED. If this succeeds, it writes its new values to memory and releases its locks *unconditionally*, as it is guaranteed to hold all of its locks and is immuned from lock stealing. However, if a transaction fails to commit — either due to a violation detected in the commit protocol, or due to the final CAS on the `statusRecord` failing — then it must release its locks *conditionally* using a CAS operation, since one or more of its locks may have been stolen by other transactions. Note that lock stealing is performed in constant time, as it involves at most 2 CAS operations per lock.

2.4 Ensuring Correctness of Atomic Operations

Readers that are familiar with the CAS operation are most likely familiar with the ABA problem [16]. A specific case of the ABA problem is avoided here due to the introduction of the local clock value that is maintained by each thread. Assume two transactions, T_1 and T_2 are executed by two threads. Assume that T_1 detects a conflict with T_2 and decides to abort T_2 and steal a lock L held by T_2. T_1 aborts T_2 using a CAS operation on its `statusRecord` variable but before it steals L,

Algorithm 1 Lock Acquisition Protocol

```
Require: lock is object of type <owner id, owner clock, version>
 1: function acquireLock(lock)
 2:     T1 ← current thread
 3:     while True do
 4:         if lock.locked = False then
 5:             obtained ← Lock(T1, lock)
 6:             if obtained = True then
 7:                 return True
 8:             end if
 9:         end if
10:         T2 ← lock.owner
11:         action ← ResolveConflict(T1, T2)
12:         if action = RETRY then
13:             continue
14:         else if action = RESTART then
15:             return False
16:         else if action = ABORT OTHER then
17:             aborted ← Abort(T2)
18:             if aborted = True then
19:                 return Steal(T1, T2, lock)
20:             else
21:                 return False
22:             end if
23:         end if
24:     end while
25: end function

26: function Lock(T, lock)
27:     exp ← <0, 0, lock.version>
28:     new ← <T.id, T.statusRecord.clock, lock.version>
29:     return CAS(lock, exp, new)
30: end function

31: function Abort(T)
32:     exp ← <RUNNING, T.statusRecord.clock>
33:     new ← <ABORTED, T.statusRecord.clock>
34:     return CAS(lock, exp, new)
35: end function

36: function Steal(T1, T2, lock)
37:     exp ← <T2.id, T2.statusRecord.clock, lock.version>
38:     new ← <T1.id, T1.statusRecord.clock, lock.version>
39:     return CAS(lock, exp, new)
40: end function
```

Algorithm 2 Commit Protocol

```
1: procedure COMMIT
2:     for all locks in write-set do
3:         acquired ← AcquireLock(lock)
4:         if acquired = False then
5:             roll-back and restart
6:         end if
7:     end for
8:     for all locks in read-set do
9:         isReadValid ← ValidateReadLock(lock)
10:        if isReadValid = False then
11:            roll-back and restart
12:        end if
13:    end for
14:    exp ← <RUNNING, statusRecord.clock>
15:    new ← <COMMITTED, statusRecord.clock>
16:    committed ← CAS(statusRecord, exp, new)
17:    if committed = True then
18:        write new values to memory
19:        release locks
20:    else
21:        roll-back and restart
22:    end if
23: end procedure
```

Algorithm 3 Example Conflict Resolution Protocol

```
1: procedure RESOLVECONFLICT(T1, T2)
2:     if T2.statusRecord = COMMITTED then
3:         return RESTART
4:     else
5:         return ABORT OTHER
6:     end if
7: end procedure
```

T_1 is switched out by the OS. T_2 detects that it has been aborted so it rolls-back and starts again, acquiring L once more. At this point, T_1 is switched back in by the OS and T_1 continues to steal L, regardless of the fact that T_2 is now active again. Both transactions now consider L to be held exclusively by them and both might attempt to write a new value to a memory address in the block that L is associated with, which might cause transaction to read an inconsistent state.

By incorporating the clock value of the lock owner into the lock variable, we ensure that a stealing transaction will only be allowed to steal a lock if the other transaction has not restarted yet. In the example above, when T_1 continues its execution after being switched back in by the OS, it will fail to CAS L since L now contains a higher clock value than was previously read by T_1.

Chapter 3

Lock Stealing for TL2

In order to evaluate our Lock Stealing approach, we implemented it as part of TL2. As the basis for our implementation we use the TL2 algorithm that is delivered with Deuce [13], an open-source Java framework for transactional memory. In this chapter we'll review the method in which we implemented the lock stealing algorithm in this environment.

In Deuce, each thread is represented by an instance of a class that implements the `Context` interface. We provide a new implementation of the `Context` interface (`org.deuce.transaction.tl2cm.Context`) that contains (besides the usual objects that are required by TL2) an `AtomicInteger` object in order to hold the `statusRecord` variable. We also implement a new `LockTable` class to support locks of 64-bits instead of the 32-bit locks used by TL2. We use larger lock objects since our locks hold 3 values: the identity of the owner of the lock or 0 if the lock is free, the local clock value of the lock owner and the version of the lock. In TL2, lock variables hold one bit to indicate if the lock is free and the rest of the lock variable is dedicated to the version of the lock.

We define a new interface, `ContentionManager`, to represent the various contention managers we implement. This interface is called whenever a conflict between two transactions is detected. In TL2 there are two cases where a transaction might conflict with another transaction: the first case may occur during the execution of the transactional code, when the transaction attempts to read a memory address that the lock which is associated with is held by another transaction. The second case may occur when the transaction attempts to acquire a lock that is held by another transaction, during its commit phase. We refer to the former as a *Read Conflict* and to the latter as a *Write Conflict*.

When a transaction detects a read conflict, it consults the contention manager by invoking its `resolveReadConflict` method. As opposed to the conflict resolution protocol presented as an example in Algorithm 3, we decided to encapsulate more logic in our contention managers by allowing them to abort a transaction themselves, instead of instructing the STM system to perform this task. The reason for this change is mostly coding convenience. Thus, in our TL2 implementation a contention manager may return one of the following actions, defined as fields of the `Action` enumeration:

- RESTART – instructs the STM system to abort the current transaction and try to execute it again.
- RETRY – instructs the STM system to try to access the memory again.
- STEAL – instructs the STM system to try to steal the lock from the owning thread. This action is only used when resolving a write conflict and after the other transaction has been aborted.
- CONTINUE – instructs the STM system to continue executing the transactional code. This action is only used when resolving a read conflict and after the other transaction has been aborted.

Recall that in TL2, locks are acquired only during the transaction's commit protocol. Thus, the contention manager may only return RESTART, RETRY or CONTINUE when resolving a read conflict. If the RESTART action is returned, the transaction rolls-back and restarts. If the RETRY action is returned the transaction reads the lock variable again in hope to find that it's free. If the CONTINUE action is returned, the transaction continues executing the transactional code. Note that the contention manager may only return CONTINUE if it aborts the other transaction or if the other transaction is already aborted when the contention manager is invoked.

When a transaction detects a write conflict, it consults the contention manager by invoking its `resolveWriteConflict` method to determine how to react. In this case, the contention manager may only return RESTART, RETRY or STEAL. If the RESTART action is returned, the transaction rolls-back and restarts. If the RETRY action is returned, the transaction attempts to acquire the lock again. If the STEAL action is returned, the transaction attempts to forcibly acquire the

lock from the owning transaction. Note that the contention manager may only return STEAL if it aborts the other transaction or the other transaction is already aborted when the contention manager is invoked.

Before returning the action to the STM system, a contention manager may take several measures, which are specific to its policy. For example, a contention manager may decide to employ back-off before returning the RETRY action, or to abort the other transaction before returning the STEAL action. We elaborate on the implementation of specific policies in Section 3.1.

3.1 Contention Managers

We implement several contention managers, which we categorize to two groups. The first group consists of some of the known contention managers presented in [19] and includes Suicide, Polite, Aggressive, Karma and Polka. The second group consists of the lock stealing contention managers and includes AggressiveLS, KarmaLS and KillPrioLS.

In order to implement the priority-based contention managers of Karma, Polka, KarmaLS, PolkaLS and KillPrioLS, we added an additional `AtomicInteger` object to each `Context` class to keep track of the thread's priority.

In the Karma and Polka based contention managers, a thread's priority is proportional to the amount of work it already did, thus favoring threads with larger transactions or threads that aborted their transaction repeatedly. In these contention managers, we increment the priority by 1 whenever a transaction performs a read operation and by 10 whenever a transaction acquired a lock. The priority is reset back to 0 when the thread successfully commits its transaction.

In the KillPrioLS contention manager, a thread's priority is proportional to the number of conflicts it so far survived, thus favoring threads that caused other threads to abort. The reasoning behind this heuristics is that a thread that caused many threads to abort their transactions should be allowed to commit its transaction as otherwise it has aborted other transactions in vain, causing a great deal of wasted work. In this contention manager we increment the priority of the thread by the priority of the thread it has now aborted. As before, the priority is reset back to 0 when the thread successfully commits its transaction. For example, assume T_1 and T_2 are transactions with a priority of p_1 and p_2 respectively. When T_1 aborts T_2, T_1's priority would be set to $p_1 + p_2 + 1$.

Algorithm 4 describes the KillPrioLS algorithm.

Algorithm 4 KillPrioLS Contention Manager Algorithm

```
1: procedure ResolveReadConflict(Context T1, Context T2)
2:     if T2.statusRecord = ABORTED then
3:         return CONTINUE
4:     end if
5:     if T1.priority ≥ T2.priority then
6:         if Abort(T2) = True then
7:             T1.priority ← T1.priority + T2.priority +1
8:             return CONTINUE
9:         end if
10:    end if
11:    return RESTART
12: end procedure

13: procedure ResolveWriteConflict(Context T1, Context T2)
14:    if T2.statusRecord = ABORTED then
15:        return STEAL
16:    end if
17:    if T1.priority ≥ T2.priority then
18:        if Abort(T2) = True then
19:            T1.priority ← T1.priority + T2.priority +1
20:            return STEAL
21:        end if
22:    end if
23:    return RESTART
24: end procedure
```

Chapter 4

Empirical Evaluation

We present here a set of benchmarks to evaluate the performance effects of lock stealing on lock-based STM algorithms. We refer to three main algorithms:

1. The TL2 implementation delivered with Deuce. The contention management strategy of this algorithm is simple: whenever a transaction detects a conflict it is restarted. Denoted as TL2.

2. Our TL2 implementation that supports the conventional contention managers, Aggressive and Karma, denoted as TL2-Aggressive and TL2-Karma respectively. We selected these contention managers as the representatives of this class of algorithms as results of Polite and Polka showed little differences in comparison to Aggressive and Karma.

3. TL2 with contention management support and Lock Stealing enabled. Denoted as TL2-AggressiveLS, TL2-KarmaLS and TL2-KillPrioLS.

Experiments were conducted on an Intel i7 920 Extreme Edition (Nehalem) 2.67 GHz processor operating four cores with two hardware threads per core, allowing for a maximum of 8 concurrent hardware threads. Results presented here are an average of 5 runs.

4.1 Overhead Analysis

Here we analyze the *fast-path* overhead of merely adding contention management hooks to the TL2 implementation. We define this as the overhead incurred when no benefit can be made from contention management. We evaluate this overhead using a red-black tree integer set benchmark.

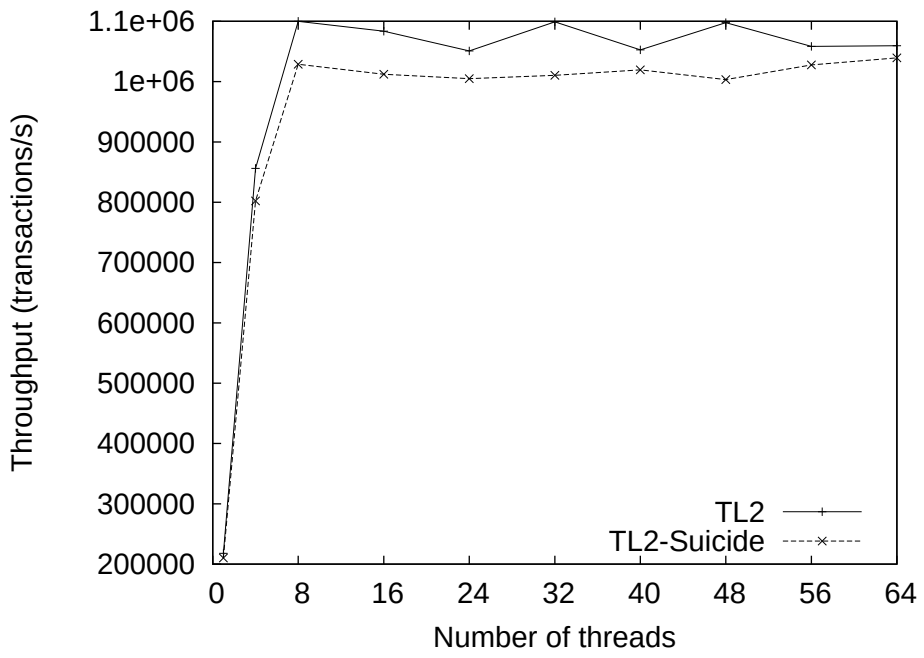

Figure 4.1: Overhead of contention management in a 64K sized red-black tree integer set with 0% update operations.

To accurately test the overhead without any performance gains due to our mechanism, we used the Suicide contention manager which performs exactly the same contention management algorithm as TL2 and configured the benchmark such that all operations are read-only, thus no contention exists between threads. Figure 4.1 shows a maximum overhead in throughput of 10%.

4.2 Integer Set Benchmark

We used the integer set benchmark that is delivered with Deuce. Each thread performs an Add, Remove or Get operation in a loop for a period of 5 seconds. We tested 3 data-structures for the set: red-black tree, linked list and skip list.

To analyze the performance effect of lock stealing, we conducted the following experiments. For each data-structure we tested a medium sized set (65,536 in red-black tree and skip list, 1,024 in linked list) and a large sized set (1,048,576 in red-black tree and skip list, 8,192 in linked list). The set was initialized to half of its maximum size. We tested two workloads: a medium contention workload (10% adds, 10% removes and 80% gets) and a high contention workload (25% adds, 25% removes and 50% gets). We also tested low contention workloads which are not presented here since

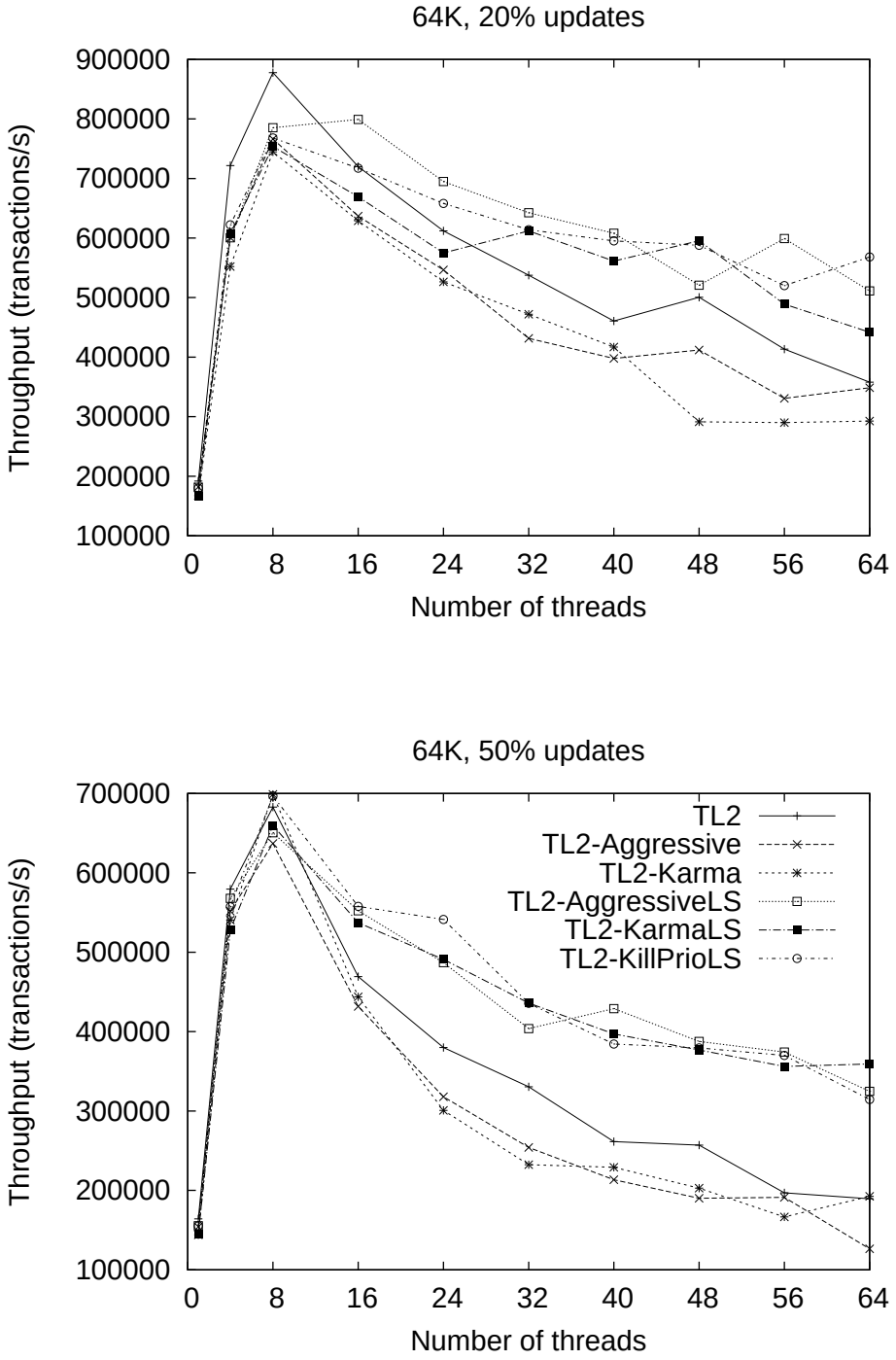

Figure 4.2: 64K Sized Red-Black Tree Integer Set

the low rate of conflicts and very little use of contention management prevented our algorithms from showing any effect on performance other than the degradation due to their overhead.

Figures 4.2 and 4.3 show the results of the red-black tree integer set. In all workloads except for

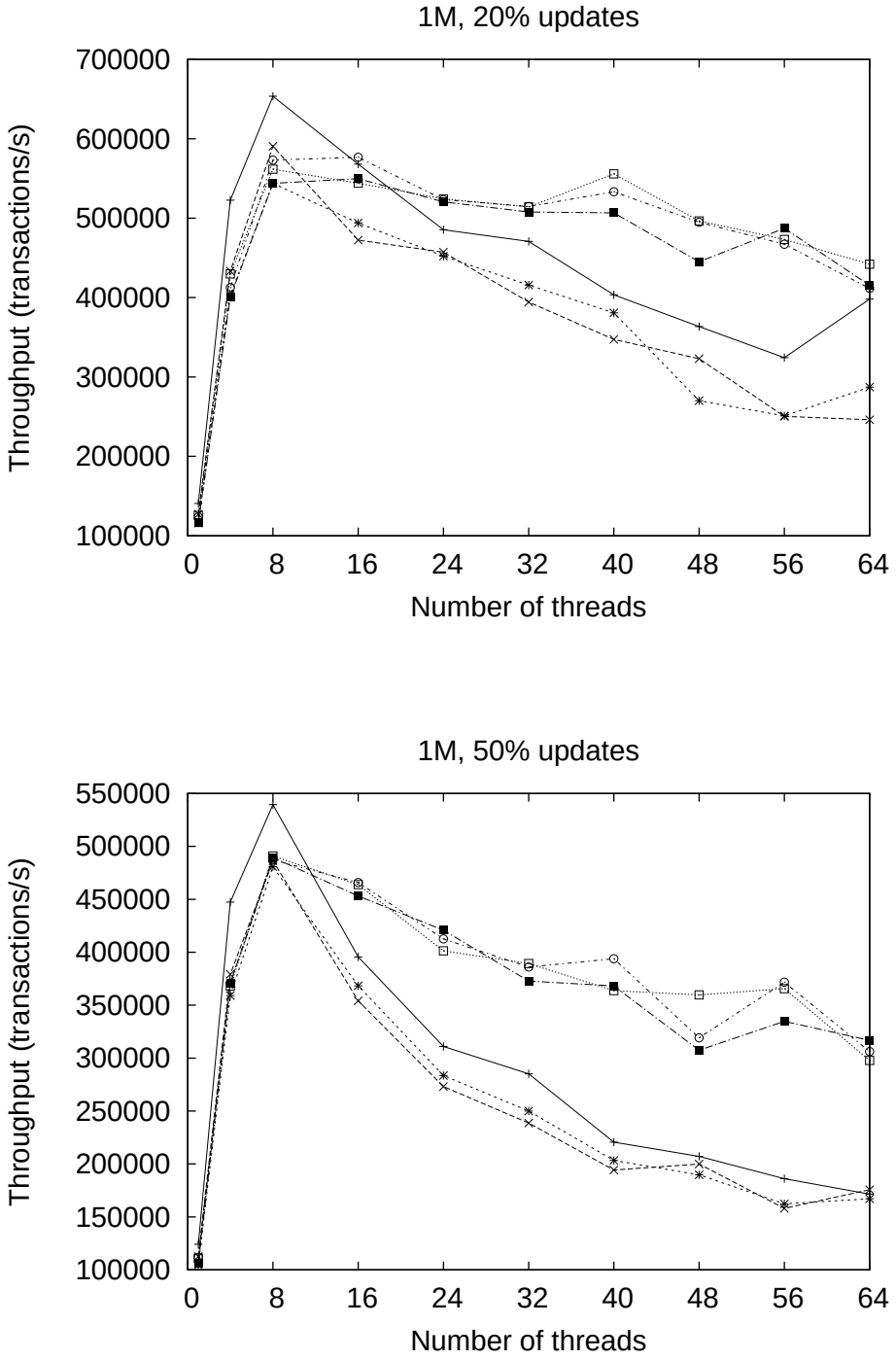

Figure 4.3: 1M Sized Red-Black Tree Integer Set

one, TL2 delivers the best peak performance. However, when the number of threads exceeds 16, the lock stealing contention managers deliver the best performance while conventional contention managers deliver the worst performance. This is due to the fact that the conventional contention

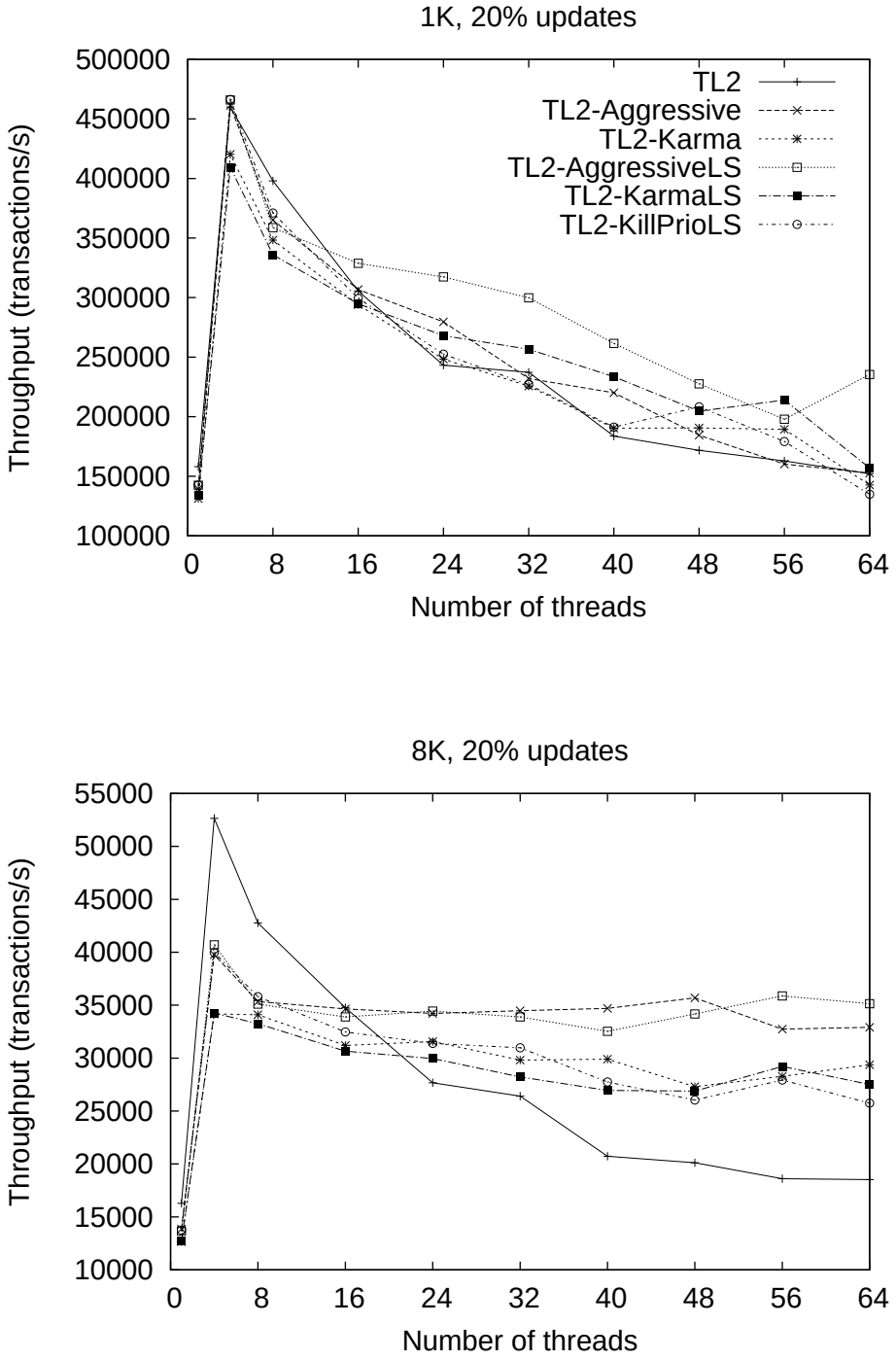

Figure 4.4: Linked List Integer Set

managers incur more overhead than TL2 but provide no relief for coping with the high rate of context switches, which cause threads that hold locks to be switched out by the OS.

Figure 4.4 shows the results of the 1K and 8K linked list integer set benchmark. TL2 performs

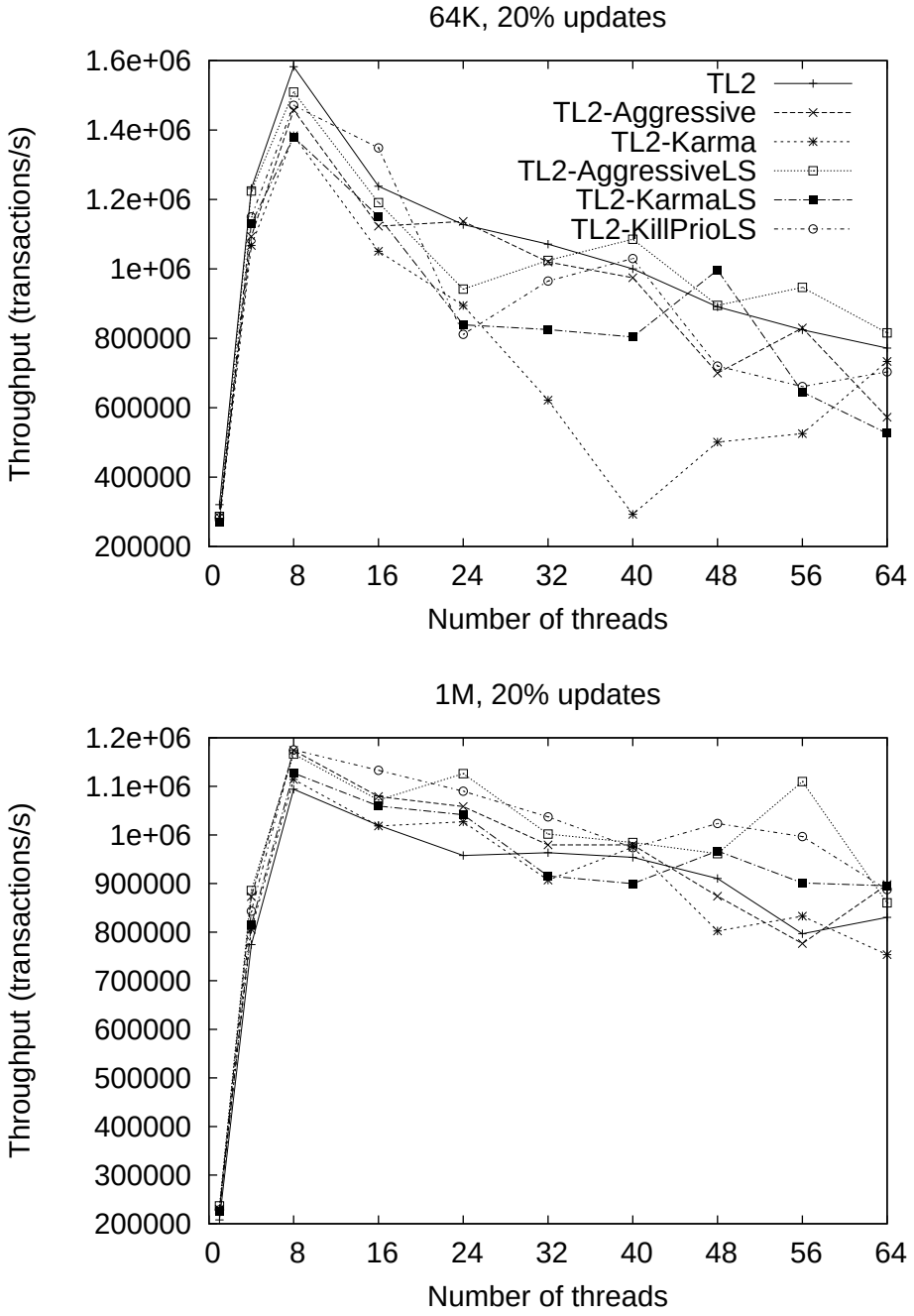

Figure 4.5: Skip List Integer Set

best when using 4 threads in both workloads, however beyond 16 threads the Aggressive and AggressiveLS contention managers deliver the best performance. Results for the high contention workloads were similar and are not shown here.

Figure 4.5 shows the results of the skip list integer set benchmark. None of the contention

managers improves or degrade performances significantly.

4.3 STAMP Applications

We used the STAMP suite of benchmark applications [2] to evaluate the performance of our approach on more elaborate test applications which were designed to simulate real world use-cases. We tested the SSCA2, KMeans, Intruder and the Vacation applications. Unfortunately, the Java port of STAMP (the original STAMP was written in C) does not yet include Bayes and Yada and the Genome and Labyrinth ports contained bugs which prevented us from using them.

4.3.1 SSCA2

The SSCA2 application constructs an efficient graph data structure using adjacency arrays and auxiliary arrays. SSCA2 shows very little contention as the graph structure is relatively large and threads rarely attempt to update the same node at the same time. In addition, SSCA2 transactions are very short, with an average read-set size of 4 and an average write-set size of 2.

Despite the low probability of conflicts and low potential for improvement in performance due to contention management, we see an improvement of 3% in throughput when using TL2-AggressiveLS in 8 threads and an improvement of 14% when using TL2-KillPrioLS in 32 threads. Figure 4.6 shows the results of the SSCA2 application. Notice the reduction in abort rate when using any of the contention managers.

4.3.2 KMeans

The KMeans application groups objects in an N-dimensional space into K clusters. Each thread processes a partition of the objects and updates the shared cluster center with its results. As K decreases, contention increases as the probability of two threads updating the same cluster simultaneously increases. We executed two workload of KMeans: a low contention workload with 40 clusters and a high contention workload with 15 clusters. KMeans transactions are of medium size with an average read-set size of 120 and an average write-set size of 25.

Figures 4.7 and 4.8 show the results of the KMeans application. When using 32 threads, KillPrioLS improves speedup by 40%-50% compared to TL2. When using 8 threads only a slight

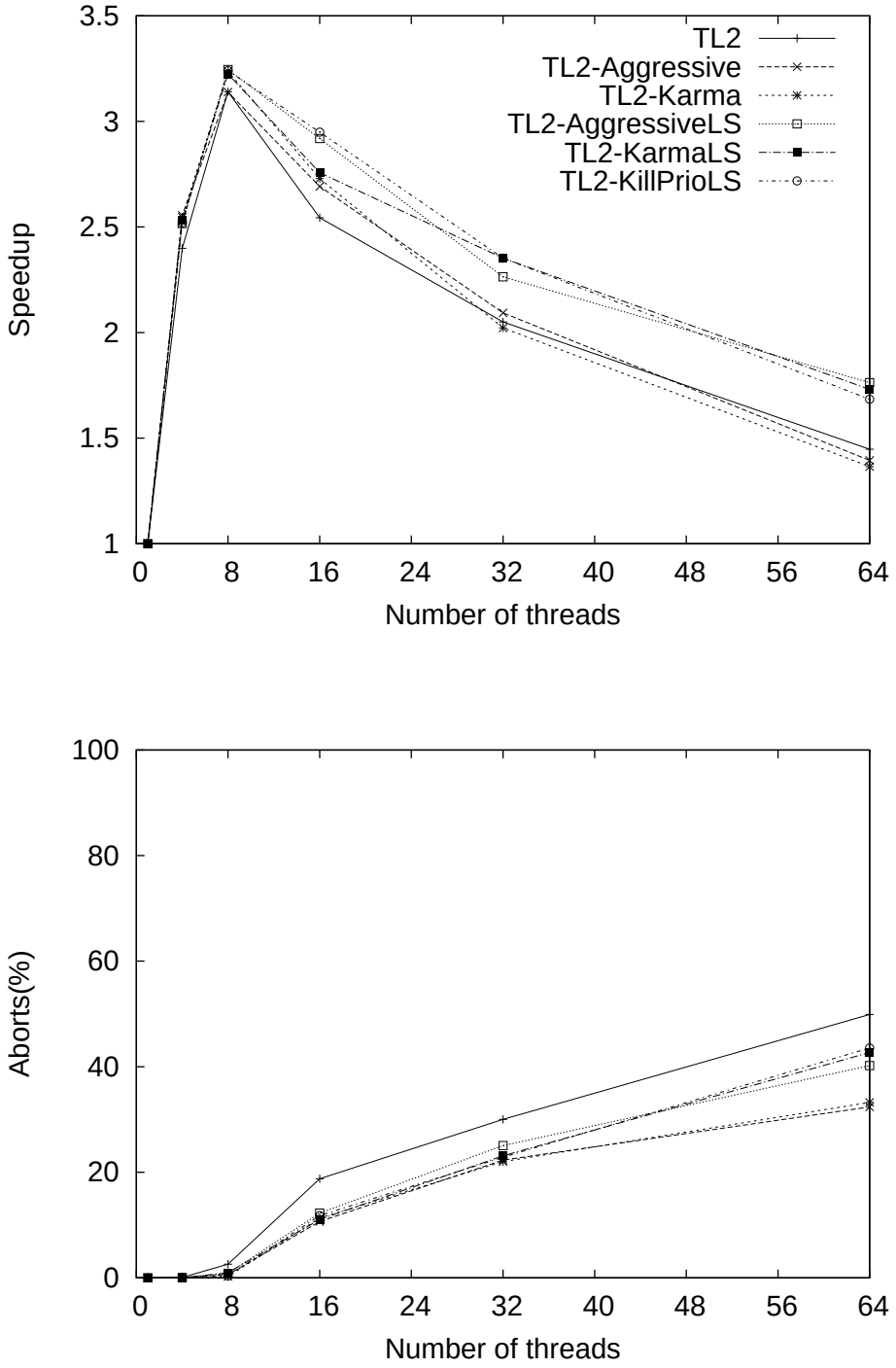

Figure 4.6: Throughput and abort rate of SSCA2

improvement by KarmaLS is shown. We did not include the Aggressive and AggressiveLS contention managers in the results as in most runs they reached a state of live-lock. In an attempt to isolate the source of the live-lock, we tested a variant of the Aggressive and AggressiveLS contention managers

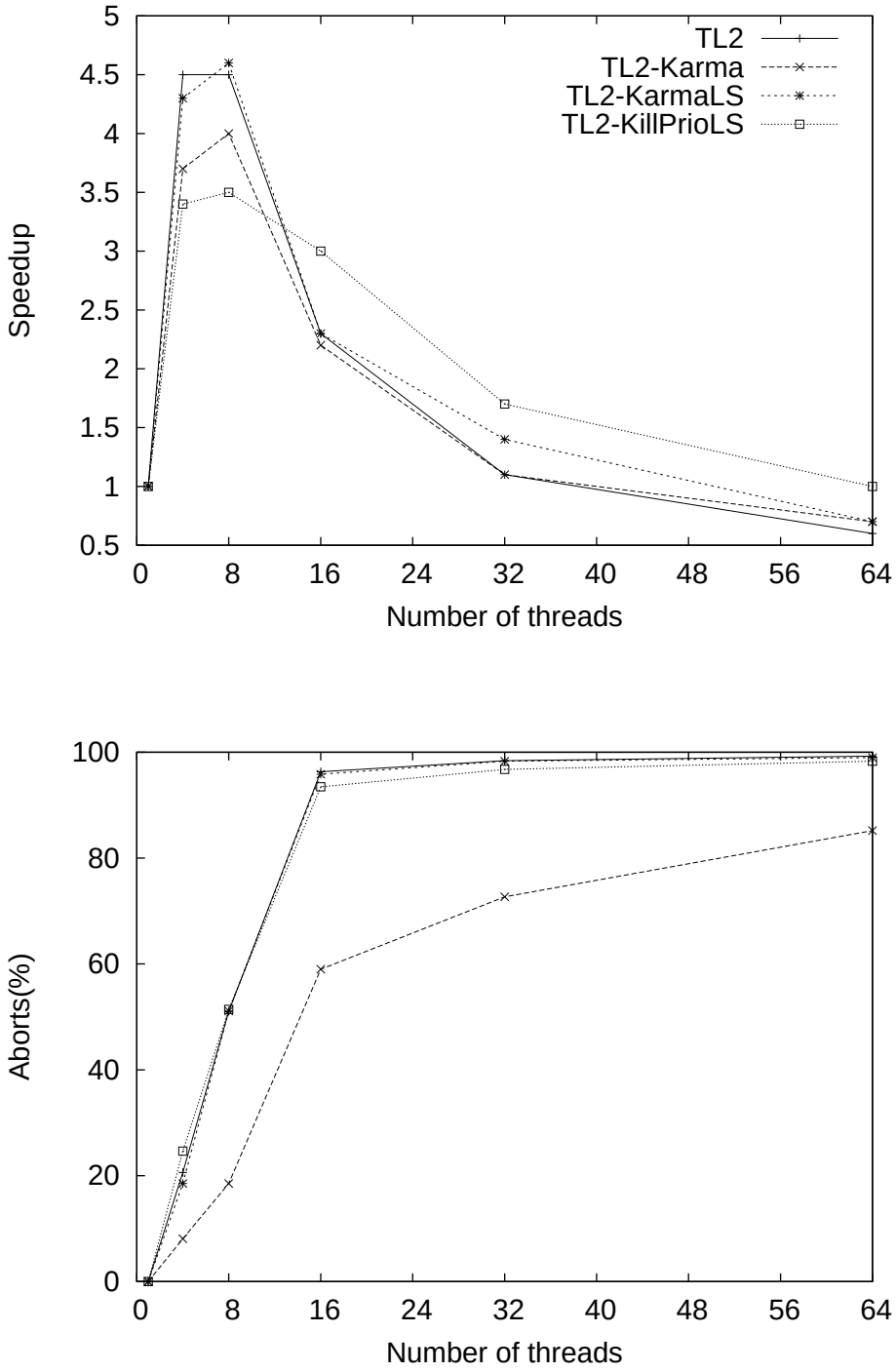

Figure 4.7: Throughput and abort rate of KMeans Low

that uses a hybrid Suicide-Aggressive policy: it automatically aborts the current transaction on read conflicts (Suicide) but still aborts the other transaction in write conflicts (Aggressive). Using this hybrid we were able to avoid the live-lock scenario. An additional attempt we made was to

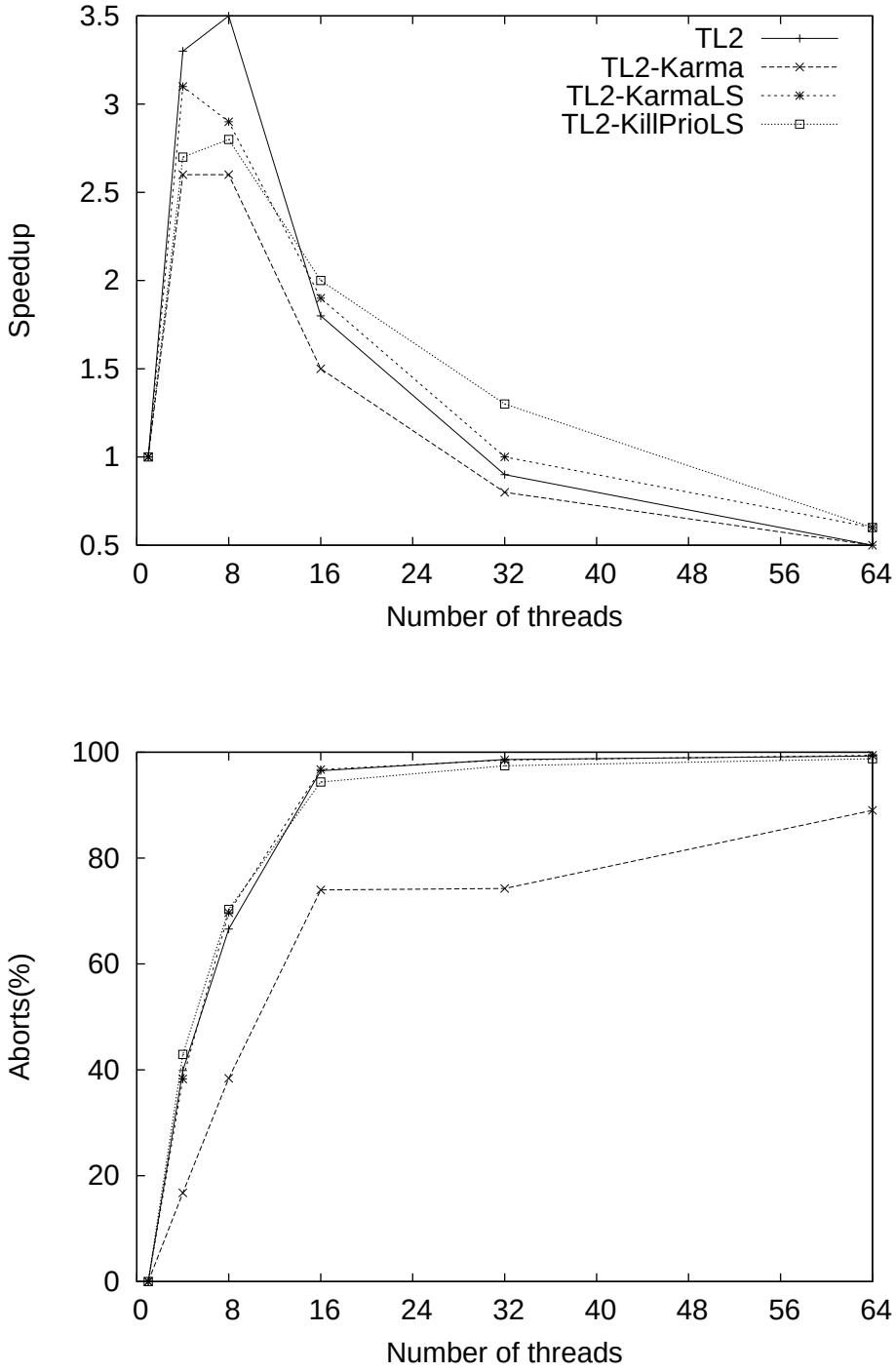

Figure 4.8: Throughput and abort rate of KMeans High

run a low contention workload of the KMeans application with a small sized input (denoted as `kmeans-low` in [2]) which also resulted in successful executions.

Since all of our contention managers (including the conventional ones) allow for a transaction to

abort another transaction in read conflicts and since we experienced the live-lock in both variants of the aggressive policy (Aggressive and AggressiveLS) we believe the live-lock is caused due to the combination of a very high abort rate (more than 90%) in conjunction with a rigid contention management policy, and not due to our Lock Stealing mechanisms.

4.3.3 Intruder

The Intruder application simulates a network intrusion detection system. Network packets are processed in parallel using two major data-structures: a FIFO queue and a dictionary (implemented using a balanced tree). Intruder transactions are relatively small with an average read-set size of 58 and an average write-set size of 6.

Figure 4.9 shows the results of the Intruder application. When using 8 threads, KarmaLS improves speedup by 20% compared to TL2 and when using 32 threads, KarmaLS improves speedup by 53%. The Aggressive contention managers constantly delivered inferior performance in this application.

4.3.4 Vacation

The Vacation application simulates a travel management system. Each thread reads and writes to a shared set of trees that keep track of customers and their reservations for various travel items. As in KMeans, we executed a low and high contention workloads. Vacation transactions are relatively large with an average read-set size of 420 and an average write-set size of 18.

Figures 4.10 and 4.11 show the results of the Vacation application. It is clear that Vacation greatly benefits from all contention managers, with a 17%-65% improvement in speedup compared to TL2.

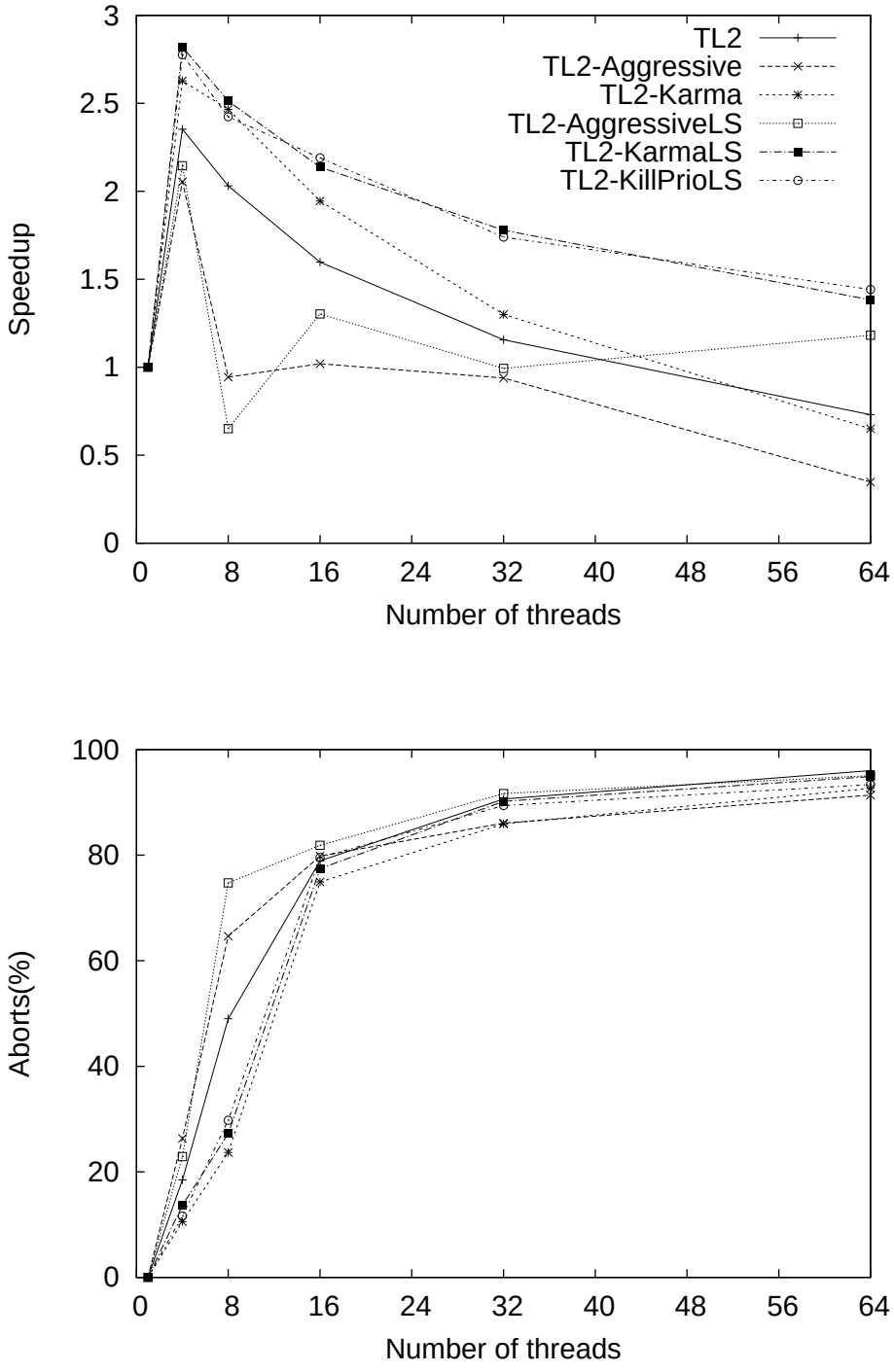

Figure 4.9: Throughput and abort rate of Intruder

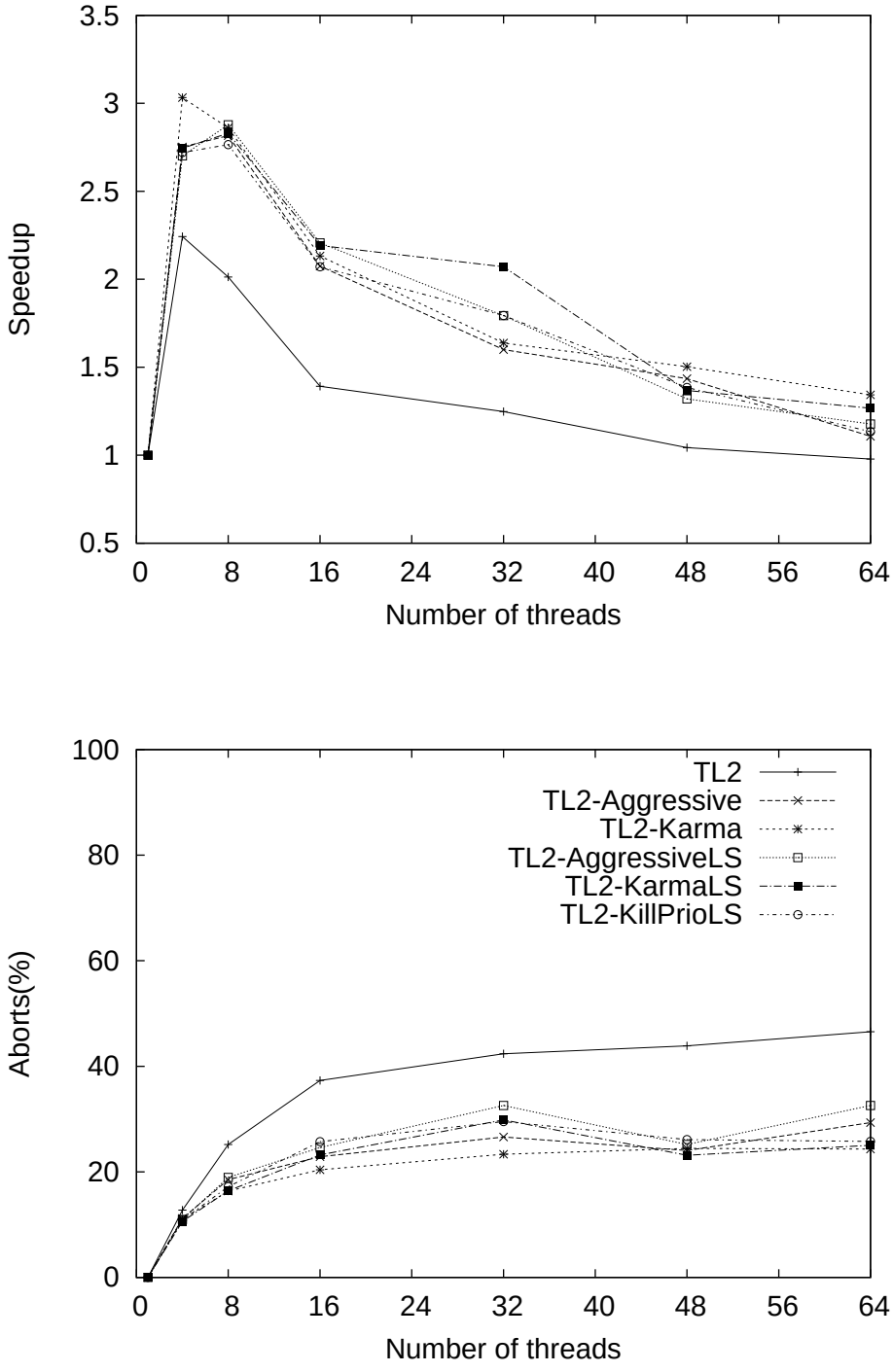

Figure 4.10: Throughput and abort rate of Vacation Low

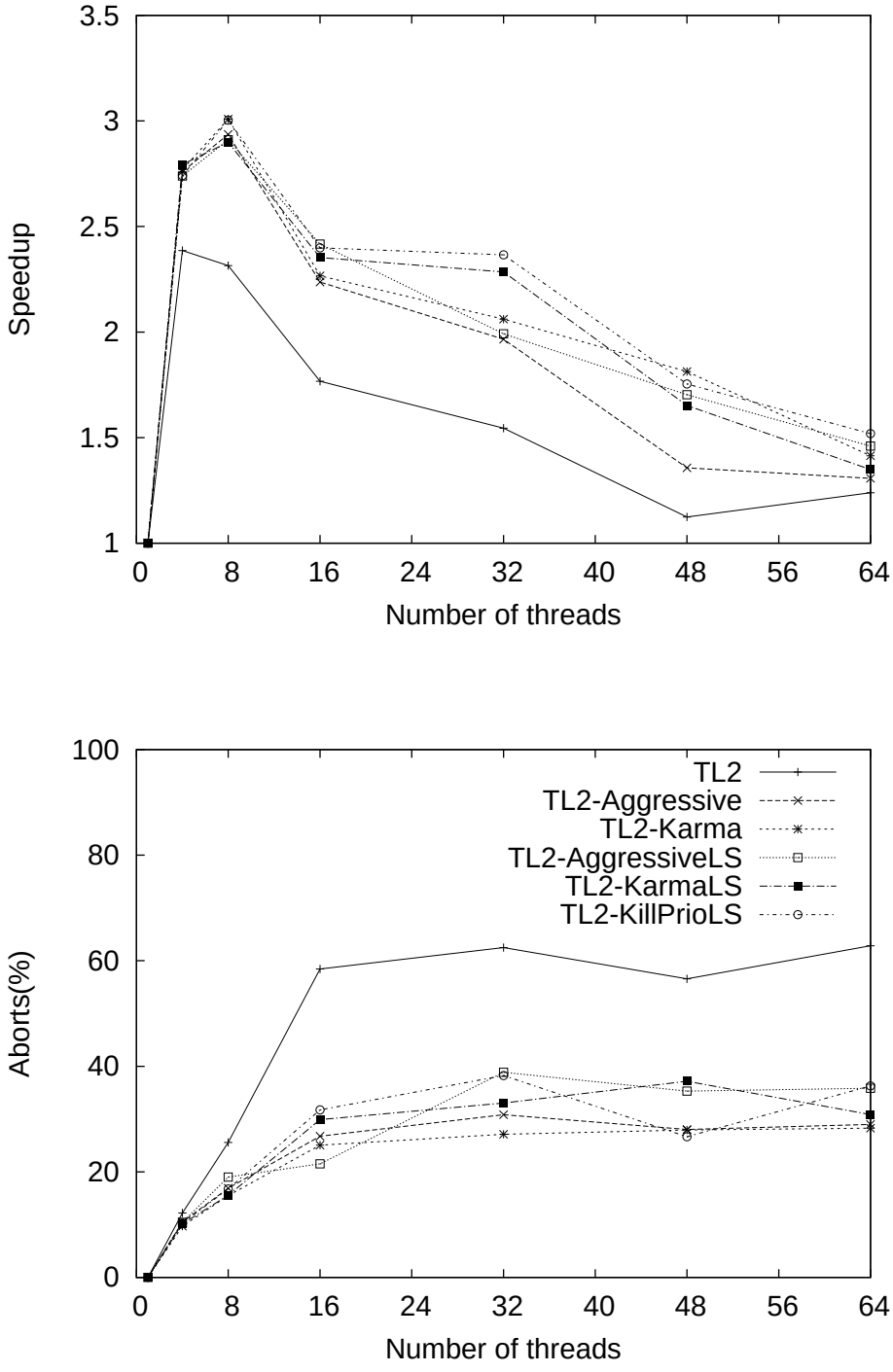

Figure 4.11: Throughput and abort rate of Vacation High

Chapter 5

Discussion

From the previous section, we observe several trends in performance of the benchmarks we tested. The most significant trend that we see is that throughput does not drop as fast compared to TL2 when using more threads than there are cores. The improvement in performance comes from the decline in abort rate - since less transactions are aborted, the throughput is higher. Abort rate is lower, since transactions are able to proceed when faced with held locks instead of aborting repeatedly. For example, in the 64K red-black tree integer set with 20% update operations as shown in Figure 4.2, the abort rate when using 32 threads in AggressiveLS is 12.4% and 31.8% in TL2. Table 5.1 summarizes the abort rates when using 32 threads in TL2 compared to the best performing contention manager.

However, a question arises – why does performance still drop? The answer is that even when using lock stealing, conflicts with transactions that are in their "Window of Un-Abortability" are still expensive, as the conflicting transactions must either wait or abort. Hence, even though optimally we would like throughput to remain at peak level this is not achievable by our mechanism.

An interesting observation from Table 5.1 is the fact that we do not see a significant improvement in abort rate for the KMeans and Intruder applications, but still observe a very significant improvement in throughput when using lock stealing. A closer look at the data shows, that when using 32 threads, both KMeans and Intruder are under extreme load, and achieve almost no speedup due to parallelism: Intruder and KMeans Low achieved a speedup of 1.1 while KMeans High achieved a speedup of 0.9. Despite the high improvement in percentage, in practice the improvement in speedup is not significant.

Table 5.1: Correlation Between Abort Rate in TL2 and Performance Improvement When Running 32 Threads

Benchmark	Abort Rate in TL2	Contention Manager	Abort Rate	Throughput Improvement
Red-Black Tree, 64K, 20%	31.8%	AggressiveLS	12.4%	+19%
Red-Black Tree, 64K, 50%	50%	KarmaLS	27.7%	+32%
Red-Black Tree, 1M, 20%	25%	KillPrioLS	10%	+9.3%
Red-Black Tree, 1M, 50%	41%	AggressiveLS	22.4%	+36%
Linked List, 1K, 20%	59%	AggressiveLS	48%	+26%
Linked List, 8K, 20%	61%	Aggressive	40%	+30%
Skip List, 64K, 20%	36%	AggressiveLS	30.3%	-3%
Skip List, 1M, 20%	27%	KillPrioLS	19.3%	+7%
Benchmark	**Abort Rate in TL2**	**Contention Manager**	**Abort Rate**	**Speedup Improvement**
SSCA2	30%	KillPrioLS	23%	+14%
KMeans Low	98.4%	KillPrioLS	96.7%	+54%
KMeans High	98.6%	KillPrioLS	97.4%	+44%
Intruder	90%	KarmaLS	90%	+53%
Vacation Low	42.3%	KarmaLS	30%	+66%
Vacation High	62.5%	KillPrioLS	38.2%	+53%

A second trend that we see in some of the benchmarks is improvement in peak throughput. This usually occurred when using as many threads as there are cores, but not always. When analyzing the relationship between peak throughput and abort rate we see that lock stealing improves throughput if there is a significant place for improvement, i.e. a relatively high abort rate in TL2. For example, Lock Stealing does not improve peak throughput in the red-black tree benchmark since its abort rate in TL2 ranges from 1.5% to 8% (depending on the workload). In contrast, in the Vacation application, the abort rates in TL2 range from 12.2% to 25.2% and a noticeable improvement in peak throughput is observed. Table 5.2 summarizes the correlation between the abort rate in TL2 when peak performance is achieved and the improvement by the best performing contention manager. Note that for Vacation High we specify two peaks, as TL2 peaked in 4 threads and the Lock Stealing contention managers peaked in 8 threads.

Table 5.2 shows another interesting point – in almost every case where performance improvement is not achieved, the best performing contention manager was Aggressive or AggressiveLS. The

Table 5.2: Correlation Between Abort Rate in TL2 and Peak Performance Improvement

Benchmark	Threads	Abort Rate in TL2	Contention Manager	Peak Throughput Improvement
Red-Black Tree, 64K, 20%	8	1.5%	AggressiveLS	-11%
Red-Black Tree, 64K, 50%	8	8%	KillPrioLS	+2%
Red-Black Tree, 1M, 20%	8	1.1%	Aggressive	-10%
Red-Black Tree, 1M, 50%	8	6.1%	AggressiveLS	-9%
Linked List, 1K, 20%	4	18.6%	KillPrioLS	+1%
Linked List, 8K, 20%	4	20.4%	AggressiveLS	-23%
Skip List, 64K, 20%	8	1.5%	AggressiveLS	-5%
Skip List, 1M, 20%	8	0.3%	KillPrioLS	+7%
Benchmark	**Threads**	**Abort Rate in TL2**	**Contention Manager**	**Speedup Improvement**
SSCA2	8	2.6%	AggressiveLS	+3%
KMeans Low	8	51%	KarmaLS	+2%
KMeans High	8	66%	KarmaLS	-18%
Intruder	4	18.5%	KarmaLS	+20%
Vacation Low	4	12.7%	Karma	+35%
Vacation High	4,8	12.2%, 25.6%	Karma, KarmaLS	+17%, +30%

aggressive contention managers have the least overhead, since they require no bookkeeping from the STM system (to calculate karma for example). Thus, in cases where our approach cannot improve, the contention managers with the least overhead deliver best performance.

From the table we can see that a very low abort rate can in-fact lead to degradation in performance, as our contention management policies have no room to improve but still incur overhead.

Chapter 6

Conclusion

In this thesis we have proposed Lock Stealing, an algorithm that can enhance lazy lock-based STM algorithms. Lock stealing enables transactions to take locks from other transactions instead of waiting for the other transactions to release them. It is therefore an algorithmic solution that does not rely on external OS-specific mechanisms, and as such can be readily used in portable applications and managed runtimes like Java.

We implemented lock stealing in Java using Deuce STM as a framework and TL2 as a base STM algorithm, and evaluated it on several benchmarks. Our results show that lock stealing is effective in parallel applications that experience a medium level of contention when the number of threads is equal to or larger than the number of cores.

Bibliography

[1] Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *High Performance Embedded Architectures and Compilers*, volume 5409 of *Lecture Notes in Computer Science*, pages 4–18. Springer Berlin / Heidelberg, 2009.

[2] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of the IEEE International Symposium on Workload Characterization*, pages 35–46, Washington, DC, USA, September 2008. IEEE Computer Society.

[3] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, volume 4167 of *LNCS*, pages 194–208. Springer-Verlag, Oct 2006.

[4] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 21–33, Washington, DC, USA, 2007. IEEE Computer Society.

[5] 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.

[6] R. Ennals. Software transactional memory should not be obstruction free. Technical Report IRC-TR-06-052, Intel Research, 2006.

[7] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 237–246, New York, NY, USA, 2008. ACM.

[8] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 258–264, New York, NY, USA, 2005. ACM.

[9] T. Harris and K. Fraser. Revocable locks for non-blocking programming. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 72–82, New York, NY, USA, 2005. ACM.

[10] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 522–529, Washington, DC, USA, 2003. IEEE Computer Society.

[11] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[12] 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.

[13] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *Programmability Issues for Multi-Core Computers (MULTIPROG'10)*, January 2010.

[14] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, and G. Muller. Scheduling support for transactional memory contention management. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 79–90, New York, NY, USA, 2010. ACM.

[15] V. J. Marathe and M. Moir. Toward high performance nonblocking software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 227–236, New York, NY, USA, 2008. ACM.

[16] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51:1–26, May 1998.

[17] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 221–228, New York, NY, USA, 2007. ACM.

[18] 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 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 187–197, New York, NY, USA, 2006. ACM.

[19] W. N. Scherer, III and M. L. Scott. Contention management in dynamic software transactional memory. In *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[20] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 240–248, New York, NY, USA, 2005. ACM.

[21] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[22] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 141–150, New York, NY, USA, 2009. ACM.

[23] J.-T. Wamhoff, T. Riegel, C. Fetzer, and P. Felber. RobuSTM: A robust software transactional memory. In S. Dolev, J. Cobb, M. Fischer, and M. Yung, editors, *Stabilization, Safety, and*

Security of Distributed Systems, volume 6366 of *Lecture Notes in Computer Science*, pages 388–404. Springer Berlin / Heidelberg, 2010.