Lightweight Identification of Captured Memory for Software Transactional Memory*

Fernando Miguel Carvalho[1,2] and João Cachopo[2]

[1] DEETC, ISEL/Polytechnic Institute of Lisbon, Portugal
mcarvalho@cc.isel.ipl.pt
[2] INESC-ID Lisboa / Instituto Superior Técnico, Universidade de Lisboa, Portugal
joao.cachopo@ist.utl.pt

Abstract. Software Transactional Memory (STM) implementations typically instrument each memory access within transactions with a call to an STM barrier to ensure the correctness of the transactions. Compared to simple memory accesses, STM barriers are complex operations that add significant overhead to transactions doing many memory accesses. Thus, whereas STMs have shown good results for micro-benchmarks, where transactions are small, they often show poor performance on real-world–sized benchmarks, where transactions are more coarse-grained and, therefore, encompass more memory accesses.

In this paper, we propose a new runtime technique for lightweight identification of captured memory—LICM—for which no STM barriers are needed. Our technique is independent of the specific STM design and can be used by any STM implemented in a managed environment. We implemented it on the Deuce STM Framework, for three different STMs, and tested it across a variety of benchmarks.

Using our technique to remove useless barriers, we improved the performance of all baseline STMs for most benchmarks, with speedups of up to 27 times. Most importantly, we were able to improve the performance of some of the benchmarks, when using an STM, to values close to or better than the performance of the best lock-based approaches.

Keywords: Software Transactional Memory, Runtime Optimizations.

1 Introduction

Some researchers (e.g. [6]) question the usefulness of Software Transactional Memory (STM), because most STM implementations fail to demonstrate applicability to real-world problems: In many cases, the performance of an STM on a real-world–sized benchmark is significantly lower than the sequential version of the benchmark, or even than the version using coarse-grain locks. The loss of performance is often attributed to the *over-instrumentation* [19] made on these

* This work was supported by national funds through FCT, both under project PEst-OE/EEI/LA0021/2013 and under project PTDC/EIA-EIA/108240/2008 (the RuLAM project).

J. Kołodziej et al. (Eds.): ICA3PP 2013, Part I, LNCS 8285, pp. 15–29, 2013.

benchmarks by overzealous STM compilers that protect each and every memory access with a barrier that calls back to the STM runtime.

Thus, several researchers proposed optimization techniques to elide useless barriers—for instance, to elide barriers when accessing transaction local memory. The most effective proposals (e.g. [2], [5], [13] and [19]) decompose the STM's API in heterogeneous parts that allow the programmer to convey application-level information about the behavior of the memory locations to the instrumentation engine. Yet, this approach contrasts with one of the main advantages of an STM, which is to provide a transparent synchronization API, meaning that programmers just need to specify which operations are atomic, without knowing which data is accessed within those operations. That is the approach used by Deuce STM [14], an STM framework for the Java environment.

Afek et al. [1] added to Deuce STM a static analysis technique to enable compile-time optimizations that avoid instrumentation of memory accesses in several situations, including to transaction local memory. Yet, this approach does not accomplish the performance improvements shown by solutions based on heterogeneous APIs that were also proposed to Deuce STM [5]. In fact, static compiler analysis is often imprecise and conservative, and thus cannot remove all unnecessary barriers, because program modules are dynamically loaded, for example, and it is impossible to perform whole program compiler analysis. However, we argue that automatic approaches that keep the transparency of the STM API are better suited to the overall goal of STMs. So, in this paper, we propose to tackle this problem and find a technique based on runtime analysis that automatically and efficiently elide STM barriers for transaction local memory.

Our work is based on the proposal of Dragojevic et al. [8], which introduces the concept of *captured memory* as memory allocated inside a transaction that cannot *escape* (i.e., is *captured* by) its allocating transaction. Captured memory corresponds to newly allocated objects that did not exist before the beginning of their allocating transaction and that, therefore, are held within the transaction until its successful commit. They use the term *capture analysis* (similar to *escape* analysis) to refer to a compile- or runtime-time algorithm that determines whether a memory location is captured by a transaction or not.

Given the lack of demonstrable effectiveness of the static compiler analysis [1], here we are interested in exploring the proposal of Dragojevic et al. [8] for *runtime capture analysis*, adapt it to a managed runtime environment and make it more efficient. More specifically, the main contributions of this paper are:

- A new runtime technique for lightweight identification of captured memory—LICM—for managed environments that is independent of the underlying STM design (Section 3). Our approach is surprisingly simple, yet effective, being up to 5 times faster than the *filtering* algorithm proposed by [8] (which we briefly introduce in Section 2.2).
- We implemented the LICM in Deuce STM, which already includes some optimization techniques in its original implementation (Section 2.1). Our implementation uses a new infrastructure of enhancement transformations,

which is described in Section 4. By providing an implementation of our proposal within Deuce STM, we were able to test it with a variety of baseline STM algorithms, namely, LSA [16], TL2 [7], and JVSTM [10].
– We performed extensive experimental tests for a wide variety of benchmarks (Section 5), including real-world–sized benchmarks that are known for being specially challenging for STMs. The goal of these tests was not only to evaluate the performance of our proposal, but, more importantly, to assess the usefulness of the runtime capture analysis, thus completing the analysis of [8] about how many of the memory accesses are to captured locations. Besides the STAMP [4], we also analyze the STMBench7 [12], and the JWormBench [5], which were not included in [8].
– For the first time, in some of the more challenging benchmarks, the LICM makes STM's performance competitive with the best fine-grained lock-based approaches. Moreover, given its lightweight nature, it has almost no overhead when the benchmark presents no opportunities for optimizations.

In Section 6, we discuss related work on optimization techniques for STMs. Finally, in Section 7, we conclude and discuss some future work.

2 Past Solutions for Compiler Over-Instrumentation

A naive STM compiler translates every memory access inside a transaction into a read or a write barrier, which typically require orders of magnitude more machine cycles than a simple memory access. So, whereas the approach taken by STM compilers ensures the correctness of the whole application, it also degrades its performance significantly. In this section, we present an overview on past solutions to elide useless STM barriers.

2.1 Deuce STM Optimizations

Deuce STM is a Java-based STM framework that provides a bytecode instrumentation engine implemented with ASM [3]. Its two major goals are: (1) to be able to integrate the implementation of any synchronization technique, and, in particular, different STMs; and (2) to provide a transparent synchronization API, meaning that a programmer using it just needs to be concerned with the identification of the methods that should execute atomically. For this purpose, the programmer marks those methods with an `@Atomic` annotation and the Deuce's engine automatically synchronizes their execution using a synchronization technique that is defined by the programmer in a class that implements the `Context` interface (for more detailed information about Deuce STM see [14]).

During instrumentation, Deuce STM can perform two optimizations to suppress useless STM barriers. First, Deuce STM does not instrument accesses to `final` fields, as they cannot be modified after creation. This optimization avoids the use of STM barriers when accessing immutable fields, provided that they were correctly identified in the application code.

Second, programmers may exclude some classes from being transformed by specifying the names of the classes to be excluded via a runtime parameter (`org.deuce.exclude`). This approach, however, reduces the transparency of the Deuce API. Moreover, it has some limitations: It does not work with arrays, nor can it be used when the same class has both instances that are shared and instances that are not shared across the transaction's boundaries. So, there is no support in the original Deuce STM for identifying objects that are transaction local and it is not feasible to do it through the existent mechanisms.

2.2 Runtime Capture Analysis

Our proposal is based on the work of Dragojevic et al. [8], originally proposed for the Intel C++ STM compiler, but that we adapted to the Deuce STM.

In Algorithm 1, we show the pseudo code for a read and a write barrier in Deuce STM when using runtime capture analysis. In both cases, the barrier first checks whether the object being accessed is captured by the current transaction. If so, it accesses data directly from memory; otherwise, it executes the standard full barrier. As in Deuce STM, object fields are updated in place using the `sun.misc.Unsafe` pseudo-standard internal library.

Algorithm 1. Read and write barriers when using runtime capture analysis

```
        ▷ in the following, ref is an object, addr is the address of the field accessed
          on ref, val is the value read/written, and ctx is the transaction's context
 1: function onReadAccess(ref, val, addr, ctx)
 2:     if isCaptured(ref, ctx) then
 3:         return val    ▷ returns the field's value if the object ref is captured by ctx
 4:     else
 5:         return ctx.onReadAccess(ref, val, addr) ▷ full STM barrier has to be used
 6:     end if
 7: end function

 8: function onWriteAccess(ref, val, addr, ctx)
 9:     if isCaptured(ref, ctx) then
10:         Unsafe.putInt(ref, addr, val)              ▷ Updates the field in-place.
11:     else
12:         ctx.onWriteAccess(ref, val, addr)          ▷ full STM barrier has to be used
13:     end if
14: end function
```

The performance of this solution depends on the overhead of the capture analysis, which is made by the `isCaptured` function. So, if the potential savings from barrier elision outweighs the cost of runtime capture analysis, then the average cost of a barrier in an application will be reduced and the overall performance will be improved.

In the Dragojevic et al's original proposal the capture analysis algorithm was intertwined with the memory management process. The key idea of their

algorithm was to compare the address of the accessed object, `ref`, with the ranges of memory locations allocated by the transaction. To perform this analysis, all transactions must keep a *transaction-local allocation log* for all allocated memory.

So, the performance of the `isCaptured` function depends on the performance of the search algorithm that needs to lookup the allocation log for a specific address, which ultimately depends on the efficiency of the data structure used to implement the allocation log. In their work, they implemented and tested three different data structures: a search tree, an array, and a *filter* of memory ranges. The search tree allows insertions and removals of memory ranges and search operations to determine if an address belongs to a memory range stored in the tree. The array implementation of the log simply keeps all memory ranges allocated inside a transaction as an unsorted array. Finally, the *filtering* approach uses a hash table as a filter: When a block of memory gets allocated, all memory locations belonging to the block are hashed and the corresponding hash table entries are marked with the exact addresses of the corresponding memory locations; thus, this filtering scheme allows false negatives.

Dragojevic et al's experimental results show similar performance improvements for the three data structures,[1] peaking at 18% for 16 threads and the Vacation benchmark in a low-contention configuration.

On a managed runtime environment with automatic memory management, we do not have readily access to the memory allocation process, so that we can log which memory blocks are allocated by a transaction and, therefore, we cannot implement the capture analysis algorithm based on the search tree or the array data structures. Thus, we adapted the hash table filtering algorithm, replacing it with an `IdentityHashMap` of the JDK and we logged the references of the objects instantiated by a transaction. In our case, and contrary to the original approach, this implementation does not allow false negatives, which increases the reliability of the capture analysis, but incurs in further overhead to maintain the transaction-local allocation log. Nevertheless, using our implementation with the TL2 STM, we get a performance improvement similar to what was shown in [8]: For a low-contention configuration of the Vacation benchmark, we achieve a performance improvement of 32% at 16 threads (see Figure 1).

3 Lightweight Identification of Captured Memory

Although the implementation of the Dragojevic et al's filtering technique improves the overall performance of Deuce STM, the `isCaptured` algorithm is still much more expensive than a simple memory access: We have to calculate the `System.identityHashCode()` for the accessed object and then we have to lookup an hash table for that object.

In fact, even with this runtime capture analysis, Deuce STM still does not perform well in some of the most challenging benchmarks, such as the Vacation [4] or

[1] With the hash table performing slightly worse, 5% in the worst case, than the alternatives.

the STMBench7 [12], where transactions are more coarse-grained and, therefore, encompass more memory accesses.

We claim that is, in part, due to the relative high cost of the `isCaptured` function, and that, if we can lower that cost, we may solve the problem. To see what is the effect of removing all the STM barriers for transaction local memory in these benchmarks, we identified the classes that are instantiated inside a transaction scope, we excluded those classes from being instrumented (in the cases where that was possible without compromising the correctness), and then we measured the speedup obtained.

In Vacation, most of the transaction local objects are arrays and, therefore, we have no easy way to avoid those STM barriers in Deuce STM. On the other hand, in STMBench7 the operations traverse a complex graph of objects by using iterators over the collections that represent the connections in that graph. Typically, these iterators are transaction local and, thus, accessing them using STM barriers adds unnecessary overhead to the STMBench7's operations. To confirm this intuition, we logged the objects instantiated in the scope of a transaction and we also logged the read-set and the write-set for each operation of the STMBench7. Thus, we could identify which barriers access transaction local objects as shown in the results of Table 1. Then, we suppressed those barriers, excluding the whole class definition from being transformed and we measured the speedup for each operation.

Table 1. Barriers suppressed for each STMBench7 operation (r and w denote read and write barrier, respectively) and the corresponding speedup on the operation when we exclude the accessed classes from being instrumented. All classes, except `LargeSetImpl`, belong to the `java.util` package.

Operation Id	st1	st2	st3	st4	st5	st6	st7	st8	st9	st10	op1	op2	op3	op4	op5	op6	op7	op8
AbstractList$Itr	w	w	w		w	w	w	w	w	w						w	w	w
AbstractMap$2$1												w	w					
HashMap			rw					rw	rw	rw								
HashMap$Entry			rw					w	w	w								
HashMap$Entry []			rw					w	w	w								
HashSet			w					w	w	w								
LargeSetImpl																		
StringBuilder				rw														
TreeMap$KeyIterator	w					w						w	w					
TreeMap$ValueIterator					w													
""$AscendingSubMap												w	w					
""$EntrySetView												w	w					
""$EntryIterator												w	w					
Speedup TL2	2.5	1.3	6.1	1.1	3.4	2.4	1.4	3.5	4.4	2.9		3.1	3.1			1.7	1.9	1.9

From the results of Table 1, we can observe that there are transaction local objects for almost all of the STMBench7's operations (except for op1, op4 and op5) and the majority of their classes are related to the iterators of the `java.util` collections, which confirms our expectations that these iterators are transaction local. In the same table we can also observe a large speedup of each operation when we avoid the STM barriers that access those transaction local objects. So, based on these results, we expect that using an efficient capture analysis

technique has a great influence on the overall performance of the STMBench7 with Deuce.

In our work, we propose to make the runtime capture analysis algorithm faster by using the following approach: We label objects with unique identifiers of their creating transaction, and then check if the accessing transaction corresponds to that label, in which case we avoid the barriers. For this purpose, every transaction keeps a *fingerprint* that it uses to mark newly allocated objects, representing the objects' *owner* transaction. Thus, the `isCaptured` algorithm just needs to check if the owner of the accessed object corresponds to the transaction's fingerprint of the executing `Context`. In this case, it performs an identity comparison between the fingerprint of the accessing transaction and the owner of the accessed object, as shown in Algorithm 2.

Algorithm 2. The LICM algorithm of the `isCaptured` function

```
1: function isCaptured(ref, ctx)
2:     return ref.owner = ctx.fingerprint
3: end function
```

Every time a new top-level transaction begins, its context gets a new unique fingerprint. So, when a new object is published by the successful commit of its allocating transaction, any previously running or newly created transaction calling the `isCaptured` method for that object will return `false`, because their fingerprint cannot be the same as the fingerprint recorded on that object. At the end of the top-level transaction, we do not need to clear the context's fingerprint because a new fingerprint will be produced on the initialization of the next top-level transaction.

The generation of new fingerprints is a delicate process that must be carefully designed to avoid adding unintended overhead to either the Deuce STM engine or the underlying STM. A naive approach to identify each transaction uniquely is to use a global counter, but this approach adds unwanted synchronization among threads that we would like to avoid. In fact, to address this problem, we considered three different options for the generation of the fingerprints: (1) use a global quiescent counter; (2) use a number of type long that is assembled by combining a thread identifier with a per-thread sequence number; and (3) use a newly allocated instance of class `Object` as a fingerprint. We discarded the first option because we cannot do it simultaneously efficient and without the support of either synchronization or any atomic operation. The other two options have both benefits and costs. The second option avoids memory allocation, but it requires some mechanism to deal with the wraparound of the numbers. On the other hand, the third option avoids rollover and aliasing issues associated with counters, but it imposes additional memory management burden.

Within these options, we chose the third, because it is the simplest approach that solves all the problems as it relies on the garbage collection subsystem to provide uniqueness and the ability to recycle unused fingerprints. Furthermore,

we do not expect to see significant differences between the alternatives, given that the fingerprint is created when the transaction starts and corresponds to a very small cost of the entire transaction. According to the results presented in Figure 1, the TL2 enhanced with the LICM technique outperforms the filtering approach and can improve the performance of the baseline STM by 60%—almost twice the speedup achieved with filtering.

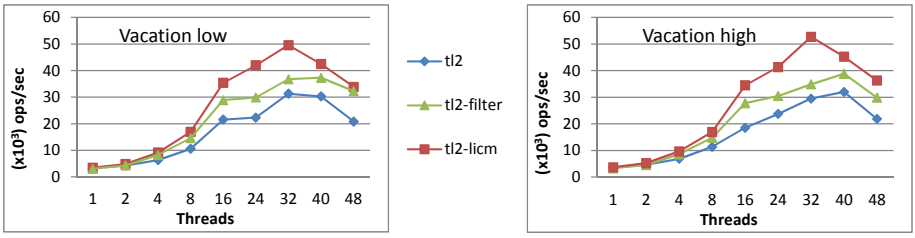

Fig. 1. The throughput for two workloads (low-contention and high-contention) of the Vacation benchmark, when using the TL2 STM. We show results for the baseline STM (*tl2*), for the STM enhanced with the filtering implementation (*tl2-filter*), and for our LICM approach (*tl2-licm*).

4 Extending Deuce STM

There are some transactional optimization techniques, such as the LICM and the multi-versioning used by JVSTM, that require a specific type system distinct from the one provided by the managed environment. Moreover, and in the particular case of the LICM, it also needs to perform additional tasks beyond the standard behaviour provided by the STM barriers. Yet, the original Deuce STM just provides extensibility in terms of the specification of the STM algorithm, but it allows neither the definition of additional behavior orthogonal to all STMs, nor any modification to the standard type system. We extended Deuce STM to support the previous requirements and we followed three major guidelines:[2] (1) to avoid changing the current Deuce STM API; (2) to guarantee retro-compatibility with existing applications and STMs for Deuce; and (3) to provide the ability to enhance any existing STM with the capture analysis technique without requiring either its recompilation or any modification to its source-code.

Extending Deuce STM with the capture analysis technique requires two main changes to the Deuce STM core structures: (1) the `Context` implementation of any STM must keep a fingerprint representing the identity of the transaction in execution and must perform the capture analysis shown in Algorithm 1; and (2) a *transactional class* (i.e., a class whose instances are accessed in a transactional scope) must have an additional field, `owner`, to store the fingerprint of the transaction that instantiates it.

[2] This adaptation of Deuce is available at `https://github.com/inesc-id-esw/deucestm/`

To support the first feature, we added a new system property, `org.deuce.filter` that enables the specification of a *filter context*—that is, a class that implements the `Context` interface and adds some functionality to any existing `Context` (using the decorator design pattern [11]). The new class `ContextFilterCapturedState` uses this approach, so that it can be applied to an existing `Context` of any STM.

To ensure that all transactional objects have a `owner` field, their classes must inherit, directly or indirectly, from the class `CapturedState`. To support this feature, we added to the Deuce STM framework a new infrastructure that allows the specification and execution of *enhancers*, which are additional transformations to the standard Deuce instrumentation. These enhancers are instances of classes implementing the interface `Enhancer` and they may be added to the Deuce engine through the system properties `org.deuce.transform.pre` and `org.deuce.transform.post`, depending on whether they should be performed before or after the standard Deuce instrumentation. Moreover, the enhancers may be combined in a chain of transformations, when more than one enhancer is specified in the same *pre* or *post* property.

5 Performance Evaluation

All the tests were performed on a machine with 4 AMD Opteron(tm) 6168 processors, each one with 12 cores, resulting in a total of 48 cores. The JVM version used was the 1.6.0_33-b03, running on Ubuntu with Linux kernel version 2.6.32.

To evaluate the performance of our approach, we used the STMBench7 [12], the STAMP [4], and the JWormBench [5] benchmarks, with the LSA [16], the TL2 [7], and the JVSTM [10] STMs, all implemented in the Deuce STM framework. In all tests we show the results for the baseline STM, for the STM with LICM support (identified by the suffix *-licm*), and for the STM with filtering support (identified by the suffix *-filter*).

Moreover, given that the STMBench7 and the JWormBench benchmarks also have a medium/fine-grained locking synchronization strategy, we also compare the performance of the lock-based approach with the STM-based approach, showing that for certain STMs, using LICM makes the performance of the STM-based approach close to (or better than) the performance of the lock-based approach. In particular, for the STMBench7 and a low number of threads, JVSTM outperforms the medium-lock approach.

5.1 STAMP Benchmarks

STAMP is a benchmark suite that attempts to represent real-world workloads in eight different applications. We tested four STAMP benchmarks: K-Means, Ssca2, Intruder, and Vacation.[3] We ran these benchmarks with the configurations

[3] The original implementation of STAMP is available as a C library and these four benchmarks are the only ones available for Java in the public repository of Deuce that are running with correct results.

proposed in [4]: For Vacation Low, "-n 256 -q 90 -u 98 -r 262144 -t 65536"; for Vacation High, "-n 256 -q 90 -u 60 -r 262144 -t 65536"; for Intruder, "-a 10 -l 128 -n 65536 -s 1"; for KMeans, "-m 15 -n 15 -t 0.00001 -i random-n65536-d32-c16.txt"; and for Ssca2, "-s 13 -i 1.0 -u 1.0 -l 13 -p 3".

In Table 2, we show the speedup of each STM with LICM support for 1 thread and for N threads. Note that a speedup higher than 1 means that the performance improved with LICM, whereas a speedup lower than 1 means that performance decreased with LICM. The results in Table 2 show that LICM improves the performance of the baseline STMs for the majority of the evaluated benchmarks and that, when it has no benefits (due to the lack of opportunities for elision of barriers), the imposed overhead is very low.

Table 2. The speedup of each STM with LICM support for 1 thread and N threads. In the latter case we also show, between parentheses, the number of threads that reach the peak of performance, with and without the LICM support, respectively. We emphasise in bold the speedup values that are higher than 1.0.

1 thread	**Vacation** Low-contention	**Vacation** High-contention	**Intruder**	**KMeans**	**Ssca2**
LSA	**1.2**	**1.2**	**1.4**	0.9	1.0
TL2	**1.1**	**1.1**	**1.2**	0.9	0.9
JVSTM	**1.1**	**1.1**	**1.2**	1.0	1.0
N threads					
LSA	**7.0** (40/8)	**6.0** (40/12)	**1.7** (16/8)	1.0 (32/32)	0.9 (8/8)
TL2	**1.6** (32/32)	**1.6** (32/40)	**1.3** (16/16)	1.0 (12/24)	1.0 (8/8)
JVSTM	**1.1** (8/8)	1.0 (40/40)	**1.1** (8/8)	1.0 (4/4)	1.0 (32/32)

The speedup we observed in Intruder and Vacation agrees with the results of [8], which provide evidence for some opportunities of elision of transaction local barriers. From our analysis, Intruder instantiates an auxiliary linked list and a `byte[]`, whose barriers can be elided with our capture analysis technique. On the other hand, Vacation performs three different kinds of operations, each one including an *initialization* phase and an *execution* phase. In the initialization phase it instantiates several arrays with the arguments that should be parametrized in the operations performed by each transaction. These auxiliary arrays are transaction local and their access barriers can be suppressed through capture analysis.

Although the performance with LICM is similar for LSA and TL2, LSA shows better speedup due to scalability problems verified in the LSA when executed without the LICM—in this case we registered a high rate of aborts due to the eager ownership acquisition approach followed by LSA. For the JVSTM we do not observe the same improvement in performance because, although LICM helps to elide useless barriers for transaction local objects, they still incur in additional metadata that penalizes the corresponding memory accesses (in the case of the TL2 and the LSA, there is no in-place metadata associated with the transactional objects).

According to [8], neither K-Means nor Ssca2 access transaction local memory and, thus, in these cases there are no opportunities for eliding barriers with capture analysis. Our results are consistent with this, but still show that our technique for capture analysis has almost no overhead in performance and it just degrades the performance of up to 10% in the worst case.

5.2 STMBench7 Benchmark

LSA and JVSTM have the best performance in the STMBench7, when compared to TL2, because of their versioning approach, which allows read-only transactions to get a valid snapshot of memory and thus, they always commit successfully. Yet, LSA shows a huge scalability problem when not using capture analysis, due to the overhead of useless STM barriers when accessing transaction local objects. This happens even in the case of the read-dominated workload because most of the read-only operations use write barriers, thereby forcing the transactions to be executed as read-write transactions. The operations are classified as read-only because they do not change shared objects, but they still need to use write barriers (when not using captured analysis) because they change transaction-local objects. When this happens, LSA cannot optimize the execution of read-only transactions. Once the useless barriers are elided with LICM, LSA can already take advantage of read-only transactions and we see that it scales for an increasing number of threads, as depicted in the results of Figure 2.

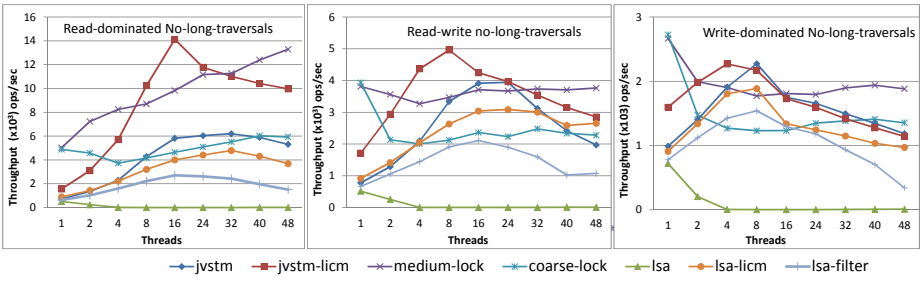

Fig. 2. The STMBench7 throughput for LSA and JVSTM, in the three available workloads, without long traversal operations. For readability reasons we omitted TL2, which is the worst of the STMs.

In the results of Figure 2 we omitted TL2, which is the STM with the worst performance. We can also observe that the performance of *LSA-licm* is between 20% and 80% better than *LSA-filter*, depending on the workload. Even though LSA-licm performs better, its results are still far from the results obtained with JVSTM-licm, which is the most performant STM in the STMBench7. In fact, JVSTM-licm gets better results than the medium-lock synchronization approach for a number of threads lower than 24. In this case, JVSTM benefits from its lock-free commit algorithm and from the lazy ownership acquisition approach, in contrast to the eager approach of the LSA.

5.3 JWormBench Benchmark

In [5], we used the JWormBench benchmark to explore the effects on performance of relaxing the transparency of an STM. To that end, we extended the Deuce API with a couple of annotations that allow programmers to specify that certain objects or arrays should not be transactified. Using this approach, we got an improvement of up to 22-fold in the performance. Now, with our new LICM technique, we got similar results but without having to change the original Deuce API.

There are two major sources of unnecessary STM barriers in the JWormBench: (1) a global immutable matrix containing the world nodes (which cannot be expressed as immutable in Java), and (2) the auxiliary arrays to the worm operations. The first barriers can be suppressed by excluding the class `World` from the instrumentation of Deuce On the other hand, the second barriers will be automatically elided through our LICM technique.

In Figure 3, we show the results obtained for the JWormBench benchmark. TL2 and LSA present the same performance in both workloads of the JWormBench and, so, we show the results for LSA only. In this case *LSA-licm* performs between 2 and 5 times faster than *LSA-filter*. Unlike what happened for the STMBench7, LSA with capture analysis is always better than JVSTM in the JWormBench, because these workloads have transactions with a smaller average length and with a lower level of contention. But, most importantly, we can see that both STMs get results close to the results obtained with the fine-grained locking approach, whereas without LICM they were an order of magnitude slower. This is true for the first workload, but when the number of write operations increases too much, as in the case of the $O(n^2), NReads, NWrites$ workload, the performance of JVSTM degrades for a higher number of threads, due to the big overhead of its read-write transactions.

The major overhead of the JWormBench comes from the mathematical operations performed by each worm. When these operations perform useless STM barriers they add a significant overhead to the transactions. In fact, and according to the observations of [5], both workloads spend almost 50% of the execution time accessing transactional local arrays through unnecessary STM barriers. Furthermore, this situation increases too much the average length of the transactions and, therefore, increases the rate of aborted transactions. In those circumstances all STMs incur in huge overheads and substantially decrease the overall throughput.

6 Related Work

Compiler over-instrumentation is one of the main reasons for the STM overheads and an obstacle to the use of STMs in real-world–sized applications. The use of unnecessary barriers on transaction-local memory access has a huge contribution to this behavior and in the past few years several solutions have been proposed to mitigate this problem.

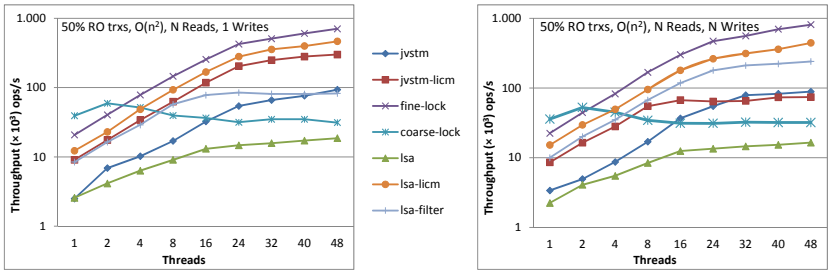

Fig. 3. The JWormBench throughput for LSA, JVSTM, and locks, for two different wokloads. Note that the vertical axes use a logarithmic scale.

One of the first contributions of Harris et al. [13] proposed a direct access STM with a new decomposed interface that is used in the translation of the atomic blocks and is exposed to the compiler, giving new opportunities for optimization. Another approach, proposed by Yoo et al. [19], is to use a new `tm_waiver` annotation to mark a function or block that should not be instrumented by the compiler for memory access—*waivered* code. Likewise, Ni et al. [15] propose that programmers have the responsibility of declaring which functions could avoid the instrumentation through the use of the annotation `tm_pure`. The same approach has been followed in managed runtime environments, such as the work of Beckman et al. [2], which proposes the use of access permissions, via Java annotations, that can be applied to references to affect the behavior of the object pointed by that reference. Carvalho et al. [5] also proposed the use of Java annotations to identify the object fields and arrays that could be accessed directly, avoiding the STM barriers.

Contrary to these approaches that involve the programmer and, thus, reduce the transparency of the STM approach, the work of Riegel et al. [17] propose to tune the behavior of the STM for individual data partitions. Their approach relies on compiler data structure analysis (DSA) to identify the partitions of an application, which may be thread-local or transaction-local. The work of Dragojevic et al. [8] propose a technique for automatic capture analysis. They provide this feature at runtime and also in the compiler using pointer analysis, which determines whether a pointer points to memory allocated inside the current transaction. Similar optimizations also appear in Wang et al. [18], and Eddon and Herlihy [9], which apply fully interprocedural analyses to discover thread-local data.

Our work builds on the work of Dragojevic et al, by proposing a lightweight technique for the runtime identification of captured memory for managed environments. A key aspect for the effectiveness of our approach is that it is performed at runtime (albeit with very low overheads). In contrast with this, Afek et al. [1] integrated static analysis in Deuce STM to eliminate redundant read and write operations in transactional methods, including accesses to transaction local-data. Yet, the results presented in their work are far from the speedups shown with our approach.

7 Conclusions and Future Work

STMs are often criticized for introducing unacceptable overhead when compared with either the sequential version or a lock-based version of any realistic benchmark. Our experience in testing STMs with several realistic benchmarks, however, is that the problem stems from having instrumentation on memory locations that are not actually shared among transactions.

Several techniques have been proposed to elide useless STM barriers in programs automatically instrumented by STM compilers. From our analysis, the main contributions in this field follow three distinct approaches: (1) runtime capture analysis; (2) compiler static analysis to elide redundant operations; and (3) decomposition of the STM APIs to allow programmers to convey the knowledge about which blocks of instructions or memory locations should not be instrumented. The latter approach is more efficient and has shown bigger improvements in the performance of the STMs, but has the inconvenient of reducing the transparency of the STMs APIs. Yet, to the extent of our knowledge, none of the previous solutions demonstrated performance improvements with the same magnitude of the results that we present here for the STMBench7 and Vacation benchmarks.

Our approach can solve one of the major bottlenecks that reduces the performance in many realistic applications and simultaneously preserve the transparency of an STM API, as shown with its implementation in the Deuce STM framework. By adding a minor overhead in memory space to all transactional objects (the reference to its owner), we get a huge speedup in the Vacation and the STMBench7 benchmarks. In fact, for the first time in the case of STMBench7, we were able to get better performance with an STM than with the medium-grain lock strategy. Moreover, integrating LICM in a managed runtime may further reduce the overhead of our approach and provide a significant boost in the usage of STMs.

References

1. Afek, Y., Korland, G., Zilberstein, A.: Lowering STM overhead with static analysis. In: Cooper, K., Mellor-Crummey, J., Sarkar, V. (eds.) LCPC 2010. LNCS, vol. 6548, pp. 31–45. Springer, Heidelberg (2011)
2. Beckman, N.E., Kim, Y.P., Stork, S., Aldrich, J.: Reducing STM overhead with access permissions. In: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming, IWACO 2009. ACM (2009)
3. Binder, W., Hulaas, J., Moret, P.: Advanced Java bytecode instrumentation. In: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, PPPJ 2007. ACM (2007)
4. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC 2008: Proceedings of The IEEE International Symposium on Workload Characterization (2008)
5. Carvalho, F.M., Cachopo, J.: STM with transparent API considered harmful. In: Xiang, Y., Cuzzocrea, A., Hobbs, M., Zhou, W. (eds.) ICA3PP 2011, Part I. LNCS, vol. 7016, pp. 326–337. Springer, Heidelberg (2011)

6. Cascaval, C., Blundell, C., Michael, M., Cain, H.W., Wu, P., Chiras, S., Chatterjee, S.: Software transactional memory: Why is it only a research toy? Queue 6(5) (September 2008)
7. Dice, D., Shalev, O., Shavit, N.N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
8. Dragojevic, A., Ni, Y., Adl-Tabatabai, A.-R.: Optimizing transactions for captured memory. In: Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2009. ACM (2009)
9. Eddon, G., Herlihy, M.P.: Language support and compiler optimizations for STM and transactional boosting. In: Janowski, T., Mohanty, H. (eds.) ICDCIT 2007. LNCS, vol. 4882, pp. 209–224. Springer, Heidelberg (2007)
10. Fernandes, S.M., Cachopo, J.: Lock-free and scalable multi-version software transactional memory. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP 2011. ACM (2011)
11. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.M.: Design patterns: Abstraction and reuse of object-oriented design. In: Wang, J. (ed.) ECOOP 1993. LNCS, vol. 707, pp. 406–431. Springer, Heidelberg (1993)
12. Guerraoui, R., Kapalka, M., Vitek, J.: STMBench7: A benchmark for software transactional memory. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys 2007. ACM (2007)
13. Harris, T., Plesko, M., Shinnar, A., Tarditi, D.: Optimizing memory transactions. In: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2006. ACM (2006)
14. Korland, G., Shavit, N., Felber, P.: Noninvasive concudrrency with Java STM. In: Electronic Proceedings of the Workshop on Programmability Issues for Multi-Core Computers. MULTIPROG (March 2010)
15. Ni, Y., Welc, A., Adl-Tabatabai, A., Bach, M., Berkowits, S., Cownie, J., Geva, R., Kozhukow, S., Narayanaswamy, R., Olivier, J., Preis, S., Saha, B., Tal, X., Tian, A.: Design and implementation of transactional constructs for C/C++. In: Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications, OOPSLA 2008 (2008)
16. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 284–298. Springer, Heidelberg (2006)
17. Riegel, T., Fetzer, C., Felber, P.: Automatic data partitioning in software transactional memories. In: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2008. ACM (2008)
18. Wang, C., Chen, W., Wu, Y., Saha, B., Adl-Tabatabai, A.: Code generation and optimization for transactional memory constructs in an unmanaged language. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO 2007 (2007)
19. Yoo, R.M., Ni, Y., Welc, A., Saha, B., Adl-Tabatabai, A.-R., Lee, H.-H.S.: Kicking the tires of software transactional memory: why the going gets tough. In: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2008. ACM (2008)