STM with Transparent API Considered Harmful

Fernando Miguel Carvalho[1,2,⋆] and Joao Cachopo[2]

[1] DEETC, ISEL/Polytechnic Institute of Lisbon, Portugal
mcarvalho@cc.isel.ipl.pt
[2] INESC-ID/Technical University of Lisbon, Portugal
joao.cachopo@ist.utl.pt

Abstract. One of the key selling points of Software Transactional Memory (STM) systems is that they simplify the development of concurrent programs, because programmers do not have to be concerned with which objects are accessed concurrently. Instead, they just have to say which operations are to be executed atomically. Yet, one of the consequences of making an STM completely transparent to the programmer is that it may incur in large overheads.

In this paper, we describe a port to Java of the WormBench benchmark, and use it to explore the effects on performance of relaxing the transparency of an STM. To that end, we implemented, in a well known STM framework (Deuce), a couple of annotations that allow programmers to specify that certain objects or fields of objects should not be transactified. Our results show that we get an improvement of up to 22-fold in the performance of the benchmark when we tell the STM framework which data is not transactional, and that the performance of the improved version is as good as or better than a fine-grained lock-based approach.

Keywords: Transactional Memory, Benchmark, Performance.

1 Introduction

The lack of realistic benchmarks is one of the factors that has been hampering the development, testing, and acceptance of Software Transactional Memory (STM) systems. Many of the developments made on STMs are evaluated on micro-benchmarks [8], [10], [14] and [16], which are fairly often criticized in some publications, such as [7]) that question the usefulness of STMs, given their lack of demonstrable applicability to real-world problems.

On the other hand, applying STMs to larger, more realistic benchmarks, such as the STMBench7 [12] and the Lee-TM [2] benchmarks, typically shows very large overheads when compared to the single-threaded sequential version of the benchmark. These overheads are often attributed to the over-instrumentation made on these benchmarks by overzealous STM engines that protect each and every memory access with a barrier that calls back to the STM runtime engine [9].

⋆ This work was supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds and by the RuLAM project (PTDC/EIA-EIA/108240/2008).

In this paper we tackle both of these problems. First, we describe a port that we made of the WormBench benchmark [18], from C# to Java. This port extends the original benchmark in several ways, making it more useful as a testbed for evaluating STMs. Moreover, our port, which we called JWormBench, was designed to be easily extensible and to allow easy integration with different STMs.

The second contribution of this paper addresses the second problem by exploring an extension of Deuce [15] that allows programmers to add annotations to their programs specifying that certain objects or fields of objects are not shared among threads, and, therefore, do not need to be instrumented in the same way as are shared data under the control of the STM, which incur in large overheads.

This contrasts with the generally accepted idea that STMs should be completely transparent, meaning that programmers just need to specify which operations are atomic, without knowing which data is accessed within those operations. That is the approach taken by Deuce, which provides a simple API based on an `@Atomic` annotation to mark methods that must have a transactional behavior.

Whereas ideally that would be the expected API for an STM, in practice that proves lead to unacceptable overheads. So, in this paper we claim that STMs with transparent APIs are harmful, and that, instead, programmers should have some degree of control on what gets transactified. To show the benefits of this, we used the JWormBench to evaluate the performance of Deuce with and without programmer annotations and show a speedup of up to 22-fold, making the optimized version on par with a very fine-grained lock-based scheme.

In the following section we identify the set of characteristics that we would like to have in a good benchmark for an STM system, and discuss to what extent some of the existing benchmarks satisfy those requirements. Then, in Section 3, we describe in more detail the WormBench benchmark, and the main differences introduced in our port of that benchmark to Java, which we called JWormBench. This new benchmark was used to evaluate the performance of Deuce, and in Section 4 we describe in which tasks the JWormBench operations instrumented by Deuce spend most of their time and how the new proposed API for Deuce can reduce some of that time. Section 5 describes the tested configurations of JWormBench and presents a performance evaluation. Section 6 provides a discussion on alternative approaches to the new API proposed for Deuce. Finally, in Section 7, we conclude with a discussion of our approach and results.

2 Benchmarks for STMs

Benchmarks are essential to test and to compare transactional memory (TM) systems. But, as realistic benchmarks are scarce, TM implementers often resort to micro-benchmarks, which are typically too simple to test their systems properly, leading to fair skepticism about the relevance of their results and the applicability of their approaches. Thus, the TM research community is in dire need of good, realistic benchmarks. But, what makes a benchmark good?

Harmanci et al. [13] distinguish two kinds of experimental evaluations for transactional memory (TM) systems: performance evaluation and semantics evaluation (debugging/testing/verification). Naturally, a good benchmark should allow both types of evaluation. On the other hand, Ansari et al. [2] argue that a TM benchmark should have as desirable features: large amounts of potential parallelism; several types of transactions; complex contention; and transactions with a wide range of durations (transaction length) and amount of data accesses (transaction size).

To the above requirements, we add that a good benchmark should: (1) be flexible enough to allow the integration of new synchronization mechanisms without requiring changes to its source-code; (2) provide a synchronization mechanism based on fine-grained locking approach, which may present a good performance and serves as a reference to achieve by other synchronizations mechanisms and (3) provide a verification test to validate the semantics of the STM and if the overall consistency of the benchmark was not broken by any erroneous synchronization.

To address the lack of realistic benchmarks for TM systems, Guerraoui et al. [12] introduced the STMBench7 benchmark: a benchmark for performance evaluation implemented in Java that models a realistic large scale CAD/CAM application. Typically, STMBench7 operations focus on object manipulation and there are no tasks that apply mathematical functions with different degrees of complexity as exist in other benchmarks, such as STAMP [5] and WormBench [18]. Furthermore, it does not provide a correctness test that is able to verify if an execution has produced correct results.

STAMP is a benchmark suite that attempts to represent real-world workloads in eight different applications. Unlike STMBench7, the STAMP applications are configurable and allow us to vary the level of contention, size of transactions, the percentage of writes, among other parameters. But this benchmark has several drawbacks also. First, not all applications make semantic evaluations of the tested STMs. Second, it is not easy to integrate STAMP applications with some STM algorithms, such as DSTM[14] and JVSTM[4], because it requires us to modify its source code and change the type of all memory locations accessed by a transaction.

In the Java world, LeeTM [2] is an alternative to STMBench7 and it has many of the desirable properties of an STM benchmark: it is based on a real-world application and provides a wide range of transaction durations and sizes. The LeeTM also provides a verifier that validates the correct consistency of the final data structure. One of the limitations of the LeeTM benchmark, however, is that it does not allow extending it to new kinds of operations and research variations of its contention scenarios. Furthermore, there is no possibility of varying the read/write ratio of the benchmark, because all of the transactions write something, unlike most applications.

WormBench [18] is a configurable transactional C# application that was designed to evaluate the performance and correctness of TM systems. The idea behind WormBench is inspired in the Snake game, but in this case the snakes are *worms* moving and performing *worm operations* in a *shared world*. The WormBench shares all the benefits of LeeTM, such as providing a verifier algorithm (correctness test) and a wide range of transaction durations and sizes. But, unlike LeeTM, it applies a broad diversity of mathematical operations and it is able to extend to new ones. Moreover WormBench is totally configurable with regard to: the percentage of update operations; the kind of operations and proportion between them; the maximum execution time or number of iterations; the contention level and synchronization strategy. In addition, the WormBench benchmark has a low complexity domain model (compared to STMBench7), making it easy to understand, and has a simple API. Still, as shown in [18], despite this simplicity, the WormBench benchmark can still reproduce STAMP workloads with the same characteristics.

3 JWormBench: A Port of WormBench to Java

The main data structures in the WormBench include *worms* formed by a *body* and a *head*, moving in a shared *world* — matrix of *nodes*. Each *node* has an integer *value* and the total sum of the values of all world's nodes is the *world's state*. For read-only workloads, the world's state should remain unchanged by the execution of the benchmark.

The *worms* perform *worm operations* on the *nodes* under the *worms*'s *head*. In the WormBench application a *worm* object is associated with one thread and is initialized with a stream of *worm operations* and *movements* that will be performed by that thread during the execution of a workload. The tasks that should be performed atomically are annotated with macros that delimit the beginning and the end of the atomic block. Then, each synchronization mechanism should translate those macros to invocations to the corresponding synchronization API.

The WormBench implementation provides 13 types of *worm operations*, which may be grouped in the following categories:

- *read-only* — *Sum*, *Average*, *Median*, *Minimum*, and *Maximum*. Each of these operations reads all the nodes under the worm's head, corresponding to *head's size*^{2} nodes;
- *n-reads-1-write* — *Replace<read-only> With<read-only>*. Each of these operations combines two of the read-only operations described in the previous item: They use the value returned by the first operation to update the node returned by the second. Each operation makes *2*head's size*^{2} reads and one write, updating the *world's state*.
- *n-reads-n-writes* — *Sort* and *Transpose*. When these operations are properly synchronized with other concurrent worm operations, they preserve the *world's state*. Each operation makes the same number of read and write operations corresponding to *head's size*^{2} nodes.

The *worm operation Sort* and those ones based on the *Median* have complexity $O(n^2)$. These algorithms could be implemented with lower complexity, but it was our intention to provide operations computationally intensive.

The JWormBench adds two new features important for the research of new workloads and evaluation of STM scalability: (1) the ability to specify the proportion between different kinds of operations, and (2) the ability to set the number of worms independently of the number of threads. Furthermore, the JWormBench provides a simple API, easy to integrate with any STM implementation in Java. So, anyone may add a new synchronization mechanism (based on STM or other), implementing the appropriate abstract types and providing those implementations to JWormBench via a configuration *module*. In the same way you can also extend JWormBench with new kinds of *worm operations* without modifying the core JWormBench library. A more detailed description about extending JWormBench is provided in JwormBench's wiki, which is available with the source-code repository[6].

To increase the extensibility of JWormBench, we have designed it according to the *inversion of control* (IoC) design pattern . Then, to run a workload on JWormBench we must create an instance of the `WormBench` class and invoke the `RunBenchmark` method. But the `WormBench` class has *dependencies* to several abstract types, whose implementations in turn depend on other abstract types and so on. So we used Guice as the *dependency injection* framework[1] to automatically resolve and inject *dependencies* based on a configuration Guice *module* (a Java class that contributes configuration information — *bindings*).

This new architecture promote the implementation and easier integration of new synchronization mechanisms without the need to interfere and modify the source code of JWormBench. Also note that this modular design does not add any additional overhead to the synchronization mechanism during the execution of the workload and while it is collecting measurements. The additional levels of indirection imposed by IoC and Guice will just delay the setup and will not affect the performance analysis.

Finally the JWormBench also provides a correctness test (i.e. sanity check for the STM system) based on the results accumulated on each threads private buffer. This buffer stores the difference between the new and the old value of every node updated by a worm operation. At the end, if we subtract the accumulated differences on each thread's private buffer to the total value of all nodes, the result must be equal to the initial sum of nodes' values.

4 Annotations to Avoid Over-Instrumentation

One of the goals of Deuce is to provide a transparent API, but the approach followed in the implementation of this feature incurs in over-instrumentation. This happens because all memory locations accessed in the context of a transaction are instrumented, independently of whether those locations are private, or not, to the transaction.

To explore the overheads caused by over-instrumentation and what may be gained by having finer grained control over what to instrument, we propose to extend the Deuce API with two Java annotations — `@NoSyncField` and

[1] Available at: http://code.google.com/p/google-guice/

`@NoSyncArray` — that can be applied on fields and type declarations, respectively, to avoid over-instrumentation in certain scenarios. In Subsection 4.1 we describe those scenarios and the reasons to over-instrumentation. Then, in Subsection 4.2, we introduce the effects and behavior of the new annotations in the Deuce framework, avoiding over-instrumentation on the previously described scenarios.

4.1 Over-Instrumented Tasks

Using profiling analysis we have verified that JWormBench is instrumented by the Deuce framework in five different accesses to memory locations. Table 1 shows the time spent by each operation accessing each memory location: *Coord* — x, y *coordinates* of worm's head; *Worm* — *Worm*'s *coordinate* array; *Node* — Value of the *node*; *World* — *World*'s *node* matrix; *local arr* — *local array* that is auxiliary to the function that defines an operation.

Table 1. Unmodified version of Deuce

#	Operation	Coord	Worm	Node	World	local arr	Flow
0	Sum	680	410	400	730	0	0
1	Average	1.100	420	440	940	0	10
2	Median	820	1.970	690	1.110	44.570	490
3	Minimum	770	650	410	990	0	0
4	Maximum	790	700	320	800	0	0
11	Sort	1.680	1.310	400	2.100	37.270	210
12	Transpose	1.490	1.170	360	1.580	1.060	130

Table 2. Optimized version of Deuce

#	Operation	Coord	Worm	Node	World	local arr	Flow
0	Sum	0	0	380	0	0	0
1	Average	0	0	430	0	0	20
2	Median	0	0	750	0	0	530
3	Minimum	0	0	480	0	0	0
4	Maximum	0	0	520	0	0	0
11	Sort	0	0	630	220	0	200
12	Transpose	0	0	320	0	0	100

Tables 1 and 2: distribution of the operation execution time (in milliseconds) accessing each of the five kinds of memory locations. There is an extra column — *Flow* — that collects the time spent in the execution of the control flow of the transactions.

The results depicted in Table 1 show that *median*, *sort* and *transpose* operations are more time-consuming because they access all kinds of memory locations. On the other hand, the *median* and *sort* operations are so much slower than the others because their algorithms have complexity $O(n^2)$.

These results were collected for JWormBench with Deuce configured to use the TL2 STM[8] and with just one worker thread. The characteristics of *world* and *worms* are the same as in the workloads for performance evaluation described in section 5. As we will see, of all the memory locations depicted in Table 1, only the third location — *Node* — should be instrumented, as it is the only one that is shared and updated by concurrent threads.

In Table 2 we present the results collected for an optimized version of Deuce according to the proposal made in this paper. Comparing the results between Tables 1 and 2 we can confirm that the third memory location is the only one where time is spent executing each *worm operation* (except for *sort*, which also wastes time accessing the *node* matrix of the *world*). Finally, table 3 shows the percentage of time due to over-instrumentation, which represents between 80% and 97% of total execution time of a *worm operation*.

Table 3. Difference between the execution time of each worm operation in unmodified Deuce framework and the optimized one

Operation	Sum	Avg	Med	Min	Max	Sort	Trans
Unmod.	2.220	2.900	49.160	2.820	2.610	42.760	5.660
Optimized	380	450	1.280	480	520	1.050	420
Over-instr	83%	84%	97%	83%	80%	98%	93%

4.2 New Java Annotations for the Deuce API

Our proposal includes two Java annotations — `@NoSyncField` and `@NoSyncArray` — that should be parametrized with a value of the *enum* type — `NoSyncBehavior`. This parameter specifies the behavior of the annotated memory location: `Immutable`, `TransactionLocal` or `ThreadLocal`.

Annotating a field with `@NoSyncField(Immutable)` has a similar effect on Deuce framework to the Java `final` keyword on field declarations. Both make the Deuce framework to avoid instrumentation when accessing those fields. However the `final` keyword has another effect at the Java level, prohibiting changes to the declared field after its initialization. This behavior could be to much restrictive for memory locations that are unmodified inside transactions, but are still target of changes outside them.

Another use of `@NoSyncField` annotation is to annotate fields that are not shared among different threads. So these fields could be private to a single transaction or to a single thread. In the first case the fields should be annotated with `@NoSyncField(TransactionLocal)` meaning that the annotated fields do not need to be synchronized and therefore do not have to be instrumented. In the second case the fields should be annotated with `@NoSyncField(ThreadLocal)` meaning that the most part of the instrumentation incurred by an STM when accessing those fields could be attenuated, excepting the *undo log* that it is still required to revert the updated data if the transaction aborts.

The effects of the `@NoSyncField(TransactionLocal)` on Deuce framework are the same as the `@NoSyncField(Immutable)`, avoiding instrumentation of the annotated memory locations when they are accessed inside a transaction. They just differ under the debug mode and in the way how the Deuce framework verifies if the specified behavior is fulfilled by the transactified program.

The `@NoSyncField(ThreadLocal)` can be applied on the declaration of memory locations such as *Coord* - x, y coordinates of worm's head. These locations belong to *coordinate* objects identifying *nodes* of the *world* that are under the *worm*'s head. The x and y fields of these *coordinate* objects are updated every time a *worm* performs a movement. However, a *worm* has affinity to only one thread and this means that a *worm* is just updated by one and always the same thread. Therefore, these memory locations do not need to be completely instrumented and can be read and updated in-place avoiding the overhead of maintaining a *read set* and *write set*. However the transactions still need to keep an *undo log* where they register the original values of the updated memory locations. Then, when a transaction aborts it uses the *undo log* to revert the data that was updated.

Annotating arrays has an increased challenge in comparison with the same task applied on fields. One difficulty in achieving this goal it is to find a way to attach the intention — *array elements are immutable, or transaction local, or thread local* — to an array declaration. The Java bytecodes for arrays manipulation receives an array reference as parameter and not the declared array variable. Then there is no easy way for the compiler to know the array variable at the moment it processes a bytecode for array access.

An alternative approach is to attach our intention to the array object instead of the array variable. However this strategy postpones the decision from compile-time to run-time and will not totally eliminate the unnecessary over-instrumentation on arrays. As we cannot annotate the array type, then we adopt a different solution: to annotate the type of the array's element with the `NoSyncArray` annotation. This approach has limitations too and may seem a little bit strange because instead of annotating intentions in the array declaration we propose to do that in the declaration of the type of the array's element.

The `@NoSyncArray(Immutable)` can be applied to the type of the array's element, whose elements are immutable during the array's life cycle. Note that the `final` keyword for arrays declaration has a distinct effect from that one that is specified by the `@NoSyncArray(Immutable)`. In the case of the `final` keyword, it just avoids the declared variable (array's reference) from being modified after its initialization and it does not mean anything about the characteristics of its elements. While the `@NoSyncArray(Immutable)` declares that the elements of the array will not change inside a transaction. This behavior is verified for memory locations as those ones referred in section 4.1 for *Worm*'s *coordinate* array and *World*'s *node* matrix.

Finally we can also use `@NoSyncArray` to annotate the declaration of the type of the array's elements as: `@NoSyncArray(TransactionLocal)` or `@NoSyncArray(ThreadLocal)`, to respectively denote that the array's elements are private to a single transaction or to a single thread.

Some *worm operations*, such as *median*, *sort* and *transpose* need an auxiliary local array to perform their algorithm. However, these arrays are private to the transaction, because their references and their elements are not shared across the boundaries of the function that defines the *worm operation*. Given that, there is no concurrent access on these arrays and there is no need to synchronize those accesses.

5 Performance Evaluation

To evaluate the effect of our approach on the performance and the scalability of the JWormBench benchmark, we compared our optimized version of the Deuce framework both against the released version 1.3.0 and against its current implementation[2]. Simultaneously, we also made a comparison with a lock-free implementation of JVSTM that has recently presented a much better performance than Deuce on the STMBench7 benchmark [10]. In our analysis we also

[2] Available at: http://code.google.com/p/deuce/

include an implementation of a fine-grained lock-based *step* for JWormBench, which acquires locks for all the nodes under a worm's head in a pre-specified order to avoid deadlocks.

The testing workload has a *world*'s size of 512 nodes and 48 *worms* with a body's length of one node and the head's size varying between 2 and 16 nodes, corresponding to a number of nodes under the worm's head between 4 and 256. The length of the body affects collisions between worms and we are not interested on that behavior for this analysis. The size of the head directly affects the length of the transaction read-set and write-set, according to the description made in section 3.

In terms of *worm operations*, we have used three different configurations. In all configurations we maintain the proportion between read-write and read-only *worm operations* of 20-80%.The configurations tested are as follows:

1. Combination of *read-only* and *n-reads-1-write worm operations*, excluding operations based on the *median* operation. So, these *worm operations* are not very intensive, having complexity *O(n)*, and the write-set for all read-write transactions has a length of one.
2. Equals to the previous configuration, but including operations based on *median*. So, this configuration is heavier than the previous one, with 4 *worm operations* having complexity $O(n^2)$.
3. Combination of *read-only* and *n-reads-n-write worm operations*. Two of these operations have complexity $O(n^2)$ and the size of the write-set, for read-write transactions, is equals to the number of nodes under *worm's head* — *head's size*^{2}.

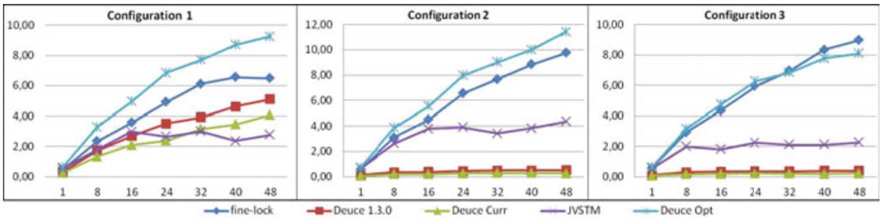

Fig. 1. Configuration 1 - *read-only* and *n-reads-1-write worm operations*, excluding *median*, Configuration 2 - *read-only* and *n-reads-1-write worm operations* and Configuration 3 - *read-only* and *n-reads-n-write worm operations*

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_22-b04`, running on Ubuntu with Linux kernel version 2.6.32.

The charts of figure 1 depict the speedup of each synchronization mechanism over sequential, non-instrumented code. These results show that Deuce 1.3.0 and Deuce's current implementation just scale for the first configuration, without the presence of heavier *worm operations*, such as *median* and *sort*. On the other hand, our optimization proposal for the Deuce framework scales in the three scenarios and presents a much better performance than the previous versions: 2 times better in the first configuration and 22 times in both configurations 2 and 3.

In comparison with the fine-grained lock approach, the optimized version of the Deuce framework has a better performance for configurations 1 and 2, but is almost equals in performance for configuration 3. The worst performance verified in configuration 3 may happen because the size of the write-set in this case is bigger than in the previous ones, increasing the number of conflicts and causing much more aborted transactions.

The JVSTM presents a similar behavior to that one shown in [10] for the STMBench7 benchmark. It scales until 16 threads and then it stabilizes. Nevertheless for configurations 2 and 3, the JVSTM has a much better performance than either Deuce 1.3.0 or Deuce's current version.

6 Related Work

More recently, Afek et al. [1] propose to solve some cases of over-instrumentation in Deuce by avoiding access barriers for thread-local memory and by using optimizations that eliminate redundant reads and writes. These optimizations pretend to avoid over-instrumentation in the same way as the annotation `@NoSyncField(TransactionLocal)`. But the remaining 5 cases that we propose to resolve with the 5 other forms of annotating memory locations, which are presented in section 4.2, are not covered by the proposal of Afek. In fact the detection of these 5 cases is not easy to implement via a static analysis as proposed by Afeck.

Our approach is also distinct from the option of an heterogeneous API as mentioned in [1], which proposes a specialized `STMReadThreadLocal()` operation beyond the generic `STMRead()`. Unlike the latter, the former assumes that the read location is thread-local and therefore avoids access barriers. But this approach is not applicable to the Deuce framework, which follows an homogeneous API.

The work of Beckman et al. [3] proposes the use of access permissions, via Java annotations, which can be applied to references, affecting the behavior of the object pointed by that reference. Besides having a different semantics, access permissions do not have direct correspondence with the Deuce framework, which is based on conflicts detection at the fields level. Unlike this work, the use of the Java annotations `@NoSyncField` and `@NoSyncArray`, proposed in our work, integrates well with how Deuce makes the instrumentation of accesses to fields.

Yoo et al [17] propose a `tm_waiver` annotation to mark a block or function that would not be instrumented by the compiler for memory access. With a unique annotation they can cover some of the cases that we propose to avoid over-instrumentation, except for thread local data. But their solution is more difficult to manage because it forces the programmer to identify all the blocks or functions that manipulate the memory locations, whose access do not need to be instrumented. In contrast in our proposal we just have to annotate the memory location on its declaration.

Our proposal also distinguishes between different behaviors of non-instrumented memory locations, allowing us to provide a debug mode in which Deuce framework validates the consistency of each location according to the behavior specified by its annotation.

7 Conclusions

STMs are often criticized for introducing unacceptable overheads 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 threads. The solution that we explore in this paper is to give the programmer some mechanisms that allow him to tell to the STM system that some memory locations are not to be manipulated transactionally. This approach reduces the transparency of the STM, which is one of its advantages over lock-based approaches, but it proves to be able to get huge benefits performance-wise. In fact, we have been able to get a 22-fold improvement on the throughput of a realistic benchmark. This result is consistent with other observations made on benchmarks that use a similar approach (e.g., the results of the JVSTM on the STMBench7 reported in [10]).

Actually, not only do we get a huge speedup when we use a less transparent API, our results show that the STM performs better or as good as a fine-grained lock-based approach, which is particularly easy to use in JWormBench, but may not be in other applications. Still, we argue that the lock-based approach is harder to develop and get right than the use of annotations to identify non-transactional memory: To implement a lock-based approach, we need not only to identify the shared resources, as in our approach, but we have to be careful about getting the locks for all of the accessed resources, and doing it in the correct order. So, even if we are losing some of the transparency of the STM approach, we believe that this may be a reasonable tradeoff between easiness of development and performance.

Finally, another contribution of this work is the JWormBench benchmark, which, despite being a port of WormBench, has some key differences from it: (1) Unlike the WormBench, which follows an STM integration approach based on macros, the JWormBench has a new solution based on *inversion of control*, *abstract factory* and *factory method* design patterns [11]; (2) the core engine of the JWormBench benchmark is deployed in a separate and independent library, whose features can be extended with other libraries; (3) unlike JWormBench, the WormBench distribution does not implement the correctness test (i.e. sanity check for the STM system) based on the results accumulated on each thread's private buffer; (4) in WormBench it is not easy to maintain the same contention scenario when varying the number of threads, while in JWormBench the number of threads is totally decoupled from the environment specification and we can maintain the same conditions along different numbers of worker threads; (5) the operations generator tool in JWormBench allows us to specify the proportion between each kind of operation, which is an essential feature to produce workloads with different ratios of update operations.

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. Ansari, M., Kotselidis, C., Jarvis, K., Luján, M., Kirkham, C., Watson, I.: Lee-TM: A non-trivial benchmark suite for transactional memory. In: Bourgeois, A.G., Zheng, S.Q. (eds.) ICA3PP 2008. LNCS, vol. 5022, pp. 196–207. Springer, Heidelberg (2008)
3. Beckman, N.E., Kim, Y.P., Stork, S., Aldrich, J.: Reducing stm overhead with access permissions. In: IWACO, pp. 2:1–2:10. ACM, New York (2009)
4. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Sci. Comput. Program. 63, 172–185 (2006)
5. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC 2008 (September 2008)
6. Carvalho, F.M. (2011), `https://github.com/inesc-id-esw/JWormBench`
7. Cascaval, C., Blundell, C., Michael, M.M., Cain, H.W., Chiras, S., Wu, P., Chatterjee, S.: Software transactional memory: why is it only a research toy? Commun. ACM 51(11), 40–46 (2008)
8. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
9. Dragojevic, A., Ni, Y., Adl-Tabatabai, A.-R.: Optimizing transactions for captured memory. In: SPAA 2009, pp. 214–222. ACM, New York (2009)
10. Fernandes, S.M., Cachopo, J.: Lock-free and scalable multi-version software transactional memory. In: PPoPP 2011, pp. 179–188. ACM, New York (2011)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: Elements of reusable object-oriented software. Addison-Wesley, Reading (1995)
12. Guerraoui, R., Kapalka, M., Vitek, J.: Stmbench7: a benchmark for software transactional memory. SIGOPS Oper. Syst. Rev. 41, 315–324 (2007)
13. Harmanci, D., Felber, P., Gramoli, V., Fetzer, C.: Tmunit: Testing transactional memories. In: TRANSACT (2009)
14. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. SIGPLAN Not. 41, 253–262 (2006)
15. Korland, G., Shavit, N., Felber, P.: Noninvasive concurrency with java stm. In: MultiProg 2010 (2010)
16. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation, pp. 284–298 (2006)
17. 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: SPAA 2008, pp. 265–274. ACM, New York (2008)
18. Zyulkyarov, F., Cristal, A., Cvijic, S., Ayguade, E., Valero, M., Unsal, O., Harris, T.: Wormbench: a configurable workload for evaluating transactional memory systems. In: MEDEA 2008, pp. 61–68. ACM, New York (2008)