STMBench7: A Benchmark for Software Transactional Memory*

Rachid Guerraoui[1] Michał Kapałka[1] Jan Vitek[2]

[1] School of Computer and Communication Sciences, EPFL
[2] S3 Lab, Department of Computer Science, Purdue University

26th September 2006

Abstract

Software transactional memory (STM) is a promising technique for controlling concurrency in modern multi-processor architectures. STM aims to be more scalable than coarse-grained locking and easier to use than fine-grained locks. However, STM implementations have yet to demonstrate that their runtime overheads are acceptable. To date, empiric evaluations of these implementations have suffered from the lack of realistic benchmarks. Measuring performance of an STM in an overly simplified setting can be at best uninformative and at worst misleading as it may steer researchers to try to optimize irrelevant aspects of their implementations.

This paper presents STMBench7: a benchmark for evaluating STM implementations. The underlying data structure consists of a set of graphs and indexes intended to be suggestive of many complex applications, e.g., CAD/CAM. A collection of operations is supported to model a wide range of workloads and concurrency patterns. Companion locking strategies serve as a baseline for STM performance comparisons.

STMBench7 strives for simplicity. Users may choose a workload, number of threads, benchmark length, as well as the possibility of structure modification and the nature of traversals of shared data structures. We illustrate the use of STMBench7 with an evaluation of a well-known software transactional memory implementation.

1 Introduction

Multi-threading is well on its way to becoming the norm in the future with the foreseen general migration to modern multi-processor systems. Whereas forking large numbers of threads is appealing for performance, controlling their concurrent interactions is tricky. The most common method for thread synchronization—using lock-based structures, like monitors—poses both efficiency and engineering problems. Coarse-grained locking is blamed for its limited scalability, whereas fine-grained locking is considered error prone.

Transactional memory [8], implemented either in software or in hardware, is an alternative to explicit locking[1] which has garnered considerable attention of late. The idea is that manipulation of shared data structures is performed within the scope of in-memory transactions. These can either commit, in which case the results of their computations become instantly visible to other threads, or abort, in which case all changes to shared state are lost. An aborted transaction may either be transparently restarted by the run-time, or the control may be handed by to the application. Deadlock and priority inversion are avoided because non-committed transactions can be aborted at any time. Furthermore, some implementations (e.g., obstruction-free ones [7, 9]) are fault-tolerant in a sense that a faulty transaction that crashes cannot cause an inconsistent state or block other transactions forever.

A Software Transaction Memory (STM) [12]

*EPFL Technical Report LPD-REPORT-2006-xxx. A part of this work has been submitted for publication.

[1] We use here the term *explicit* locking to contrast it with *implicit* locking that might underly some transactional memory implementations.

implementation guarantees atomicity and isolation of transactions through software mechanisms. These are used for undoing changes made by aborted transactions and for resolving conflicts between transactions that compete for the same shared objects. STM systems are of particular interest because they do not require change to the underlying hardware—they can be implemented either as part of the high-level language compiler, the virtual execution environment, or even as an external library. Many implementations (e.g., ASTM [9] (Java), SXM [6] (C#), RSTM [10, 2] (C++)) have been proposed. So far, there has been very little in the way of empiric evaluation of the tradeoffs in the different systems. This for two reasons: Firstly, direct comparison is difficult for systems based on different programming languages or running on customized virtual execution environments. Secondly, because there are no benchmarks that provide realistic workloads for STMs. The upshot is that all experimental evaluation to date have either relied on "toy" benchmarks based on simple data structures (e.g., lists, red-black trees), benchmarks with limited concurrency (like SPEC JVM98 or Java Grande) which were not designed for transactional memory. In most of the existing evaluations, a clear, uncontroversial, baseline is generally missing—a comparison with fine-grained and coarse-grained locking—and so it is difficult to estimate the real cost of using an STM (which is usually not negligible).

The motivation of this work is to come up with a comprehensive benchmark suite for STM implementations. More specifically, our goal is to produce a set of workloads that:

- corresponds to realistic, complex, object-oriented applications which benefit from multi-threading;
- are non-trivial to synchronize in a scalable way;
- do not depend on any particular STM technology or programming language. Concurrency can be controlled by different mechanisms at different granularities;
- are easy to use and provide results of which can be readily interpreted.

This paper presents STMBench7, a first step towards achieving that objective. Instead of starting from scratch, we considered extending a rich data structure with a long history of use for benchmarking purposes: the OO7 benchmark [4]. OO7 has been originally designed to compare various object-oriented database systems. It is not specific to any particular application, but, as shown by the authors, represents a wide variety of commercial applications including CAD, CAM or CASE systems. Like OO7, STMBench7 operates over a rich object-graph with millions of objects and many interconnections between them. There are over forty operations with various scope and complexity. This allows for simulating many different real-world scenarios and makes concurrency a non-trivial issue.

The set of operations we designed and implemented for STMBench7 is, however, significantly more involved than in OO7. Basically, OO7 was used to evaluate the performance of isolated transactions, whereas STMBench7 is aimed to consider various concurrency patterns and workloads. Furthermore, unlike in OO7, the data structure of STMBench7 is highly dynamic, which better matches the requirements of applications that allocate and deallocate memory at high rates. STMBench7 is multi-threaded and we needed to define precisely how updates to different objects performed by a single operation have to become visible to concurrent threads. STMBench7 also provides locking mechanisms that can serve as a comparison baseline for STM implementations. In its default configuration, STMBench7 comes with a coarse-grained locking strategy and a fine-grained one in order to highlight the performance and scalability tradeoffs of different strategies. In the long run we expect to provide more refined lock-based implementations.

The current implementation of STMBench7 is a little over 4000 lines of code and is available at [3]. Our version is written in Java, although, we expect to provide versions of the benchmark for other languages (such as C# or C++). STMBench7 uses standard classes from the `java.util` package. We used the new features of Java 5, such as generics and enumerations to improve the quality and readability of code. The locking strategies use the read-write locks from the `java.util.concurrent` package.

We illustrate the use of STMBench7 with an evaluation of a variant of ASTM, and we indirectly highlight the difficulty in outperforming locking strategies. Our straightforward implementation of STMBench7 using ASTM performs 2–4 orders of magnitude worse than the lock-based versions. That is because of long traversals and large objects that would need more adaptive mechanisms

than the ones ASTM uses. One way to overcome this problem would be to refactor the implementation of the data structures so that small objects are grouped and larger ones are split into smaller objects. But doing so would require significant effort and weaken the main selling point of the STM technology—namely, that it makes implementing scalable concurrent data structure easy.

Our results may be surprising for some preliminary performance evaluations have shown situations where ASTM outperforms DSTM that, in turn, scales better than coarse-grained locking strategies [9, 7]. We argue that there is actually no contradiction here. When selecting STMBench7 workloads that resemble the ones of synthetic benchmarks used so far, the ASTM-based implementation is nearly as fast as the lock-based ones, outperforming the coarse-grained locking strategy for read-dominated workload. The performance problems of ASTM are, we believe, common to many STMs that use invisible reads and object-level logging of changes made by transactions. Fortunately, some solutions to overcome these issues have already been proposed [5, 10, 11, 13].

The rest of the paper is organized as follows. We first give an overview of STMBench7. Then, we focus on its operations and concurrency aspects. We also show some experimental results that highlight the differences between the two locking strategies built in STMBench7. Finally, we illustrate the use of our benchmark by evaluating a variant of the ASTM framework.

2 Overview

As we pointed out, STMBench7 is based on the data structure underlying OO7. We had to provide, however, a new collection of operations to match the demands of concurrent applications. Basically, the implementation of STMBench7 has about 4300 lines of code of which only 2500 corresponds to the OO7 specification. In this section, we recall the OO7 benchmark and then give an overview of STMBench7, before describing its details in the next sections.

2.1 The OO7 Benchmark

The OO7 benchmark [4] has been originally designed to compare various object-oriented database systems. A precise description of OO7 can be found in its specification and the accompanying source code [1]. Here we only give a general overview of OO7 that is necessary to understand the specifics of STMBench7.

The data structure underlying OO7 is depicted in Figure 1. It consists of several *modules*, each containing a tree of assemblies. The internal nodes of the tree are called *complex assemblies* and the leaves—*base assemblies*. Each base assembly contains several *composite parts*. A composite part has a *document* assigned to it and links to a graph of *atomic parts* which are connected via *connection* objects. Each element of the data structure contains links to its parents. As a consequence, a traversal is possible both top-down and bottom-up. The many-to-many connections between base assemblies and composite parts are implemented with two bags each: one containing all composite parts belonging to a given assembly, and one containing all base assemblies a given composite part belongs to. Each document and each graph of atomic parts is associated with one composite part. On the contrary, composite parts form a design library that is shared between all base assemblies.

OO7 includes three kinds of operations: *traversals*, *queries* and *structure modifications*. Traversals go through the data structure top-down, starting from the root assembly, or bottom-up, starting from a random atomic part. Most of them access (read or update) a large subset of all shared objects. Queries generally search for a subset of objects using an index or a set. Structure modification operations create or delete a base assembly and the descendant composite parts together with their documentation objects and their graphs of atomic parts. In general, only atomic parts and documentation objects can be updated, while all others are read-only.

2.2 From OO7 to STMBench7

OO7 was designed to measure the latency of isolated operations issued to an object-oriented database system. Specific aspects of OO7 were oriented towards multi-client systems, but the benchmark was rather intended for use in low-load scenarios, where interaction between concurrent operations is not taken into account.

Our main goal in designing STMBench7 was to measure the performance (throughput and latency) of a set of operations that are interleaved by a scheduler or run in parallel, and that compete for access to shared objects. Thus, we are interested in the behavior of both the overall system

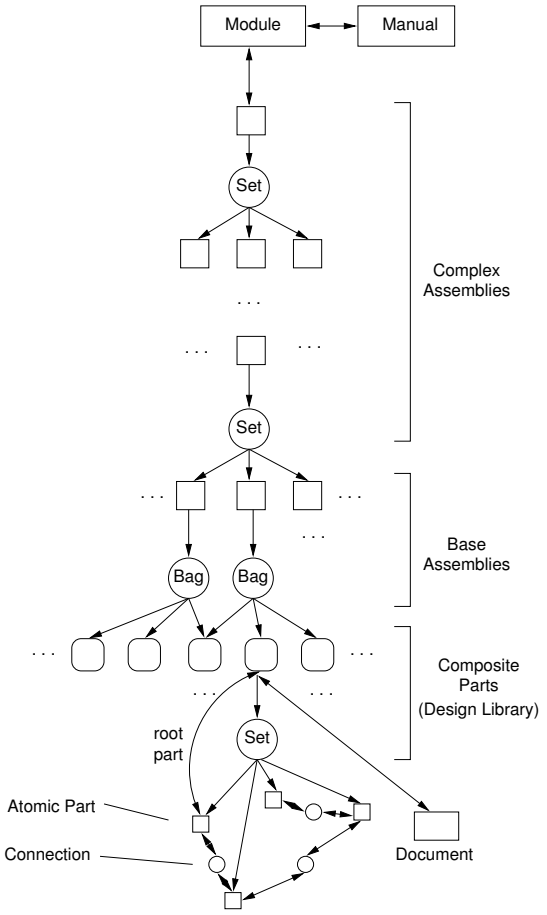


Figure 1: Overview of the basic OO7/STMBench7 data structure: *module* objects contain a tree of assemblies. Internal nodes of the tree are called *complex assemblies* and leaves are *base assemblies*. A base assembly contains several *composite parts*. A composite part has a *document* and links to a graph of *atomic parts* connected via *connection* objects. Each element contains links to its parents allowing bottom-up as well as top-down traversals.

and each individual operation, under high load and high contention. The data structure and operations of OO7 were a good starting point, but they were clearly insufficient for our purpose. Firstly, in many concurrent applications one can often find a large number of very short operations the performance of which is crucial. The traversals and queries forming the OO7 suite are mostly long and access a large number of shared objects. Secondly, the choice of the operations in OO7 makes most of the shared data structure effectively read-only. This is not very important when there is no concurrency. However, having a large number of read-only objects makes the synchronization problem unrealistically easy. Clearly, for read-dominated workloads, updates of object attributes are rare, but still the synchronization strategy, be it locking or STM-based, has to account for these rare changes that usually may appear in every part of a data structure.

	Key	*Value*
1	Atomic part ID	Atomic part
2	Atomic part build date	Atomic part
3	Composite part ID	Composite part
4	Document title	Document
5	Base assembly ID	Base assembly
6	Complex assembly ID	Complex assembly

Table 1: The list of indexes used in STMBench7.

While extending OO7, we wanted to retain the realism of its operations. However, we needed to enlarge the set of operations so that many interesting data access patterns, which often appear in concurrent programs, are tested and the related problems of synchronizing concurrent objects are faced. We give a precise description of STMBench7 operations in Section 3. We left the original data structure of OO7 almost untouched. We only removed few parts that only make sense in a database context. In particular, we removed some indexes and sets (we left the indexes listed in Table 1.) as well as indirect links between atomic parts and documents that were introduced for the sole purpose of evaluating join operations (which are quite meaningless outside the database context).

We also confined the data structure of STMBench7 to a single module. This is because multiple modules would limit the concurrency of operations and would require a much higher load to discover all the efficiency problems resulting from contention. However, we chose the "medium" size of OO7 as the base for our benchmark, which gives quite a large data structure. Namely, there are six levels of complex assemblies, having three children assemblies each, 500 composite parts altogether, each corresponding to a graph of 100000 atomic parts and at least three times as many connections between them.

STMBench7, unlike OO7, is inherently multithreaded. Therefore, additional care had to be taken so that the specification of operations is unambiguous even in presence of concurrency and contention. For example, we needed to define precisely, how updates to different objects performed

by a single operations have to become visible to concurrent threads. STMBench7 also provides two locking strategies with different granularity and complexity. We describe them precisely, together with the multi-threading issues, in Section 4.

2.3 Using STMBench7

The command-line interface of STMBench7 involves the following parameters: the length of the benchmark, the number of threads, the type of the workload (read-dominated, read-write or write-dominated) and two parameters that can independently disable long traversals and structure modification operations. The benchmark, by default, outputs the count and maximum latency numbers for each operation type and for each category of operations, as well as the total throughput. It also computes the error of the sample of randomly chosen operations, as compared to the ratios derived from the benchmark parameters. The benchmark can also optionally produce latency histograms for each operation. (See Appendix A for an STMBench7 user's guide.)

STMBench7 does not output a single number as a benchmark result. This would lead to simplistic comparisons: for some applications it is crucial to optimize the latency of long operations whereas others focus more on the throughput of short queries. Besides, interpreting a single result, computed from many others with a convoluted formula would say very little on where optimizations should actually be performed.

3 Operations, Workloads and Concurrency Patterns

STMBench7 contains 45 operations on the shared data structure. This is a large number and leaving a user full control over how often each of them is executed would be unacceptable. Therefore, we divided the operations into several categories. The benchmark assigns ratios to these categories automatically, based on the abstract description of a target application provided by a user. Then, STMBench7 operations are executed by a number of threads, in proportions that depend on the computed ratios (operations from the same category have equal ratios).

There are four main categories of STMBench7 operations:

1. Long traversals—go through all assemblies and/or all atomic parts. Some of them update documents or atomic parts. They all originate from OO7 (traversals T1–T6 and queries Q6, Q7).

2. Short traversals—traverse the structure via a randomly chosen path, starting from a module, a document or an atomic part. Some of them use indexes. One short traversal behaves differently: it iterates over all base assemblies and checks some of their descendant composite parts. Short traversals are denoted by ST1–ST10. Some of them originate in OO7 (T7, Q4 and Q5 in OO7) and some perform updates on atomic parts or documents.

3. Short operations—chose some object (or a few objects) in the structure (randomly or with some search criteria, mostly using an index) and perform an operation on the object(s) or its local neighborhood. They are denoted by OP1–OP15. Five of them originate from OO7 (Q1–Q3 and T8, T9 in OO7).

4. Structure modification operations—create or delete elements of the structure or links between elements (randomly). The operations are constrained though, so that the structure is never degenerated in a significant way. For example, the root complex assembly is always connected to all base assemblies. Also the maximum size of the structure is confined. Structure modification operations are denoted by SM1–SM8. They have no exact equivalents in OO7. A simple example of a structure modification operation is depicted in Figure 2.

We also split the STMBench7 operations into two other categories, spanning all traversals and short operations: read-only operations and update ones. Appendix B contains the full specification of STMBench7, including the description of all the operations.

A user describes a target application by providing the following information:

- Workload type: which can be read-dominated, read-write or write-dominated.
- Types of allowed operations: i.e., whether long traversals and/or structure modification operations are enabled.
- Number of concurrent threads.

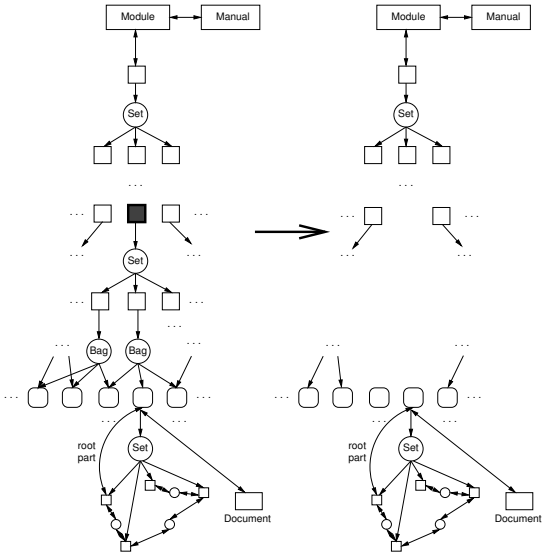


Figure 2: A structure modification operation that removes a complex assembly with its descendants

The default ratios for different operation categories are presented in Table 2. These are combined and adjusted, based on the benchmark parameters.

The operations of STMBench7 represent all the important ways the shared data structure can be accessed. There is one significant exception, though. Namely, we do not exploit concurrency patterns in which a thread must wait for results of operations performed by other threads. Thus, STMBench7 is not meant to evaluate the performance of producer-consumer-like scenarios. STMBench7 approaches the problem in a different way: it allows an operation that cannot proceed without being blocked to fail. We use this mechanism extensively, because operations lack input data and thus have to make choices randomly. For example, some operations chose an object in the structure by picking a random ID and searching an appropriate index. If the ID does not correspond to any existing object, the operations fail. Clearly, we could make them check first which IDs are available, so that they can never fail, but this would be more costly than a simple index search, which, in turn, would skew the benchmark results.

4 Multi-threading and Locking

STMBench7 runs a user-specified number of concurrent threads, all performing operations on the shared data structure. The threads are uniform in a sense that each picks its next operation randomly from the whole pool of 45 STMBench7 operations. Each thread registers locally its performance measurements. These are combined at the end of the benchmark.

There is an important question about the behavior of STMBench7 operations executed concurrently. More precisely, one has to decide whether an operation should be executed (logically) atomically or whether the updates it makes to the shared data structures can become gradually visible to all threads. The problem is difficult, because the operations of OO7, or the ones we added for STMBench7, are not tightly bound to any specific application so there is no clear semantics behind them. For real programs atomicity is not always a must—it can be weakened sometimes for efficiency reasons, when the application can take additional measures to prevent dangerous inconsistencies in the global state. We, however, have to be conservative. We thus assume that every operation is atomic, i.e., that the changes it makes to the shared data structure have to become visible instantaneously to others. It does make locking more difficult, but it also makes the lock-based version of STMBench7 have the same semantics as an STM-based one in which every operation is a single transaction.

The core code of STMBench7 does not contain any concurrency control mechanisms. This makes it possible to directly use STMBench7 with an arbitrary STM framework, without the need to remove locks and convert critical sections. Nevertheless, we do provide two locking strategies that can serve as a baseline for STM performance results, but these are provided separately and can be automatically merged with the core STMBench7 code at compile time.

The two locking strategies of STMBench7 differ in their granularity and complexity. The first, which we call "coarse-grained", uses a single read-write lock to protect the whole data structure. Clearly, it induces minimal locking overhead on operations, but limits scalability in a significant way, except for read-dominated workloads. The second strategy could be described as a pragmatic approach. It is not fully fine-grained, but its complexity (from a programmer's perspective) is similar to that of an STM-based solution. It represents what, we believe, an average software engineer would try in the first place. We call it "medium-grained". This locking strategy, in short, (1) protects each level of the data structure with a sin-

Category	Workload type		
	Read-dom.	Read-write	Write-dom.
Read-only ops	90	60	10
Update ops	10	40	90
Long Traversals	5		
Short traversals	40		
Short operations	45		
Structure mods	10		

Table 2: Default ratios for operation categories (in %).

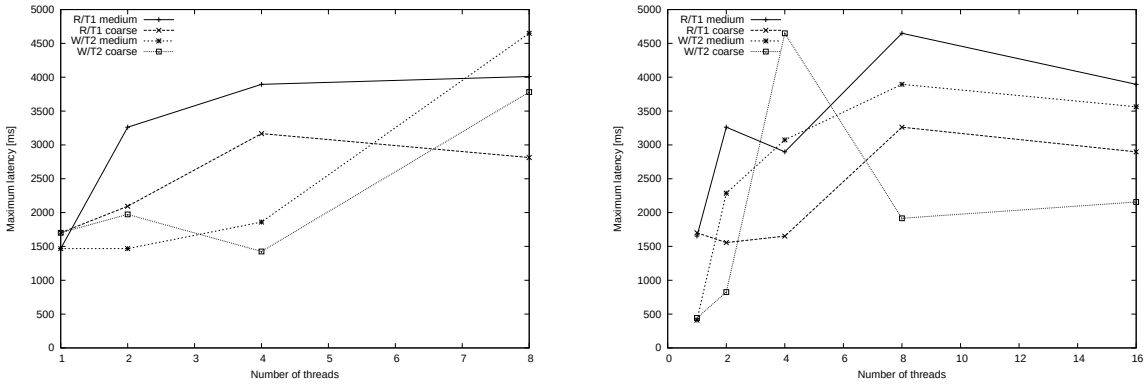


Figure 3: Comparison of coarse- and medium-grained locking strategies included in the STMBench7 suite. Maximum latency for traversal T1 (read-dominated workload) or T2b (write-dominated workload). All operations enabled. Left: 2-cpu Xeon, right: 8-cpu Sun V40z.

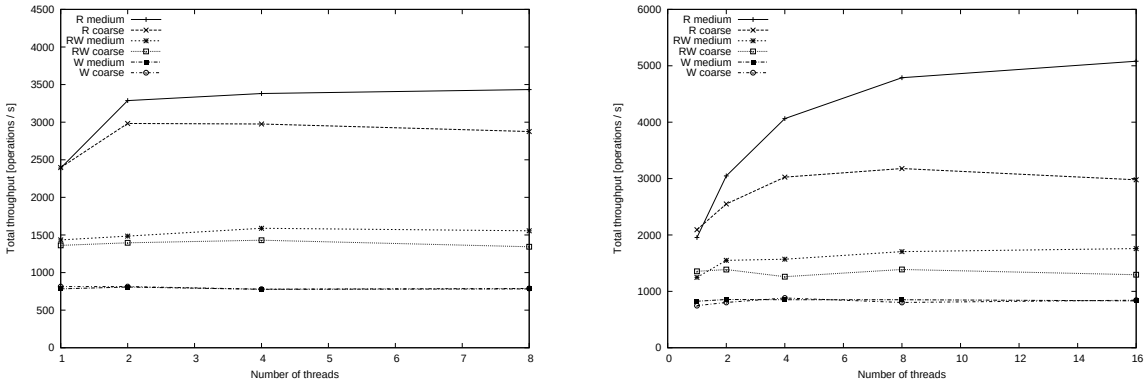


Figure 4: Comparison of coarse- and medium-grained locking strategies included in the STMBench7 suite. Total throughput with all operations except for long traversals enabled. Left: 2-cpu Xeon, right: 8-cpu Sun V40z.

gle read-write lock, and (2) makes all the structure modification operations performed in isolation (see Figure 5). More precisely, there is a single read-write lock for: (1) each level in the assembly tree, (2) all composite parts, (3) all atomic parts, (4) all documents, and (5) the manual. An additional

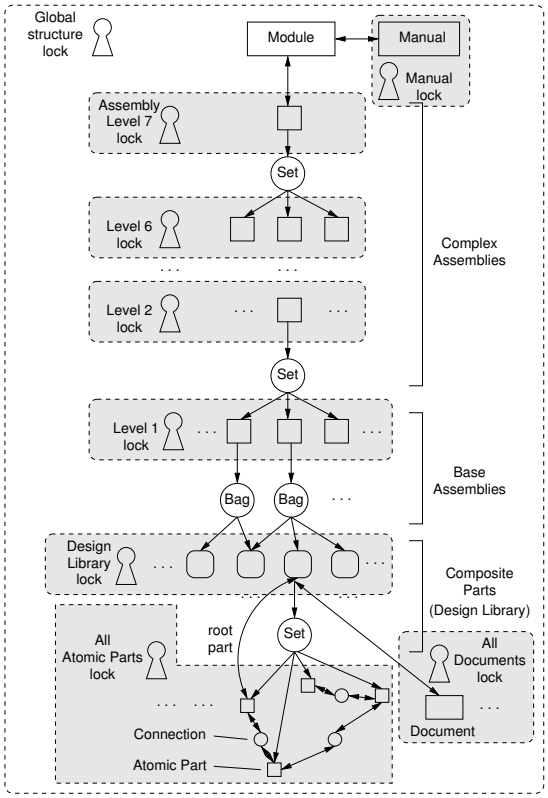


Figure 5: Medium-grained locking

read-write lock isolates structure modification operations (it is acquired in write mode by structure modification operations and in read mode by all other ones). Indexes, sets and bags do not have to be synchronized separately in this case.

A fine-grained locking could be implemented for STMBench7. It would probably make no sense to protect each atomic part with a single lock, but locking each assembly and composite part separately could result in better scalability. However, the diversity of STMBench7 operations makes the problem of fine-grained locking very difficult. That is because the data structure can be accessed in many ways and traversed in many directions. Thus, there is a need for each operation to build a list of objects it wants to access, sort the list and then acquire locks in the right order to avoid deadlocks. This, clearly, adds additional overhead which, together with the significant engineering cost, would be difficult to justify with an increase in scalability.

To illustrate the difference between the two strategies, we present here some experimental results. These were obtained on two machines: a 2-cpu Xeon and an 8-cpu Sun V40z. The maximum latency of long traversals T1 (read-only, for read-dominated workload) and T2b (updates all atomic parts, for write-dominated workload), in executions with all operations enabled, is plotted in Figure 3. The throughput results for three possible workload types, with long traversals disabled, are presented in Figure 4. Note that measuring total throughput when long traversals are allowed, as well as latency for short operations, makes little sense. That is why STMBench7 measures and outputs a variety of parameters.

Clearly, the medium-grained locking approach has a slightly larger overhead than the coarse-grained one, but it exploits the power of the multiprocessor architecture better when there are at least two concurrent threads. The scalability of medium-grained locking is hampered for write-dominated workloads, though. This is because most of the update operations and short traversals, and all structure modification operations, acquire the same locks in write mode, which means that only few can be executed in parallel. However, most of them may, at least partially, overlap with read-only operations. A finer-grained locking strategy could help here, but, as we already mentioned, implementing it efficiently would be much more complex than using an STM.

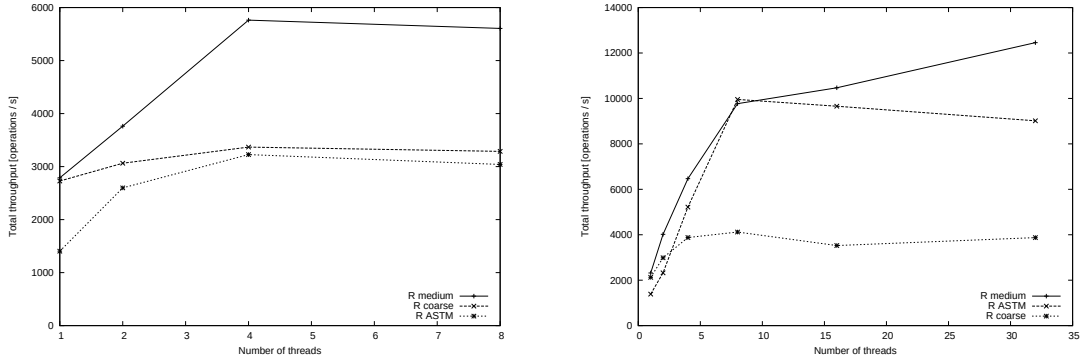
The throughput results are not surprising. However, the latency ones need some explanation, for the latency of long traversals T1 and T2b is, on average, higher for medium-grained locking than for coarse-grained one. We believe this is due to the fact that long traversals have to acquire 9 locks in the former case, and only a single lock in the latter case. Thus, with medium-grained locking a thread executing T1 or T2b has to wait more often in lock queues.

5 Illustration: ASTM

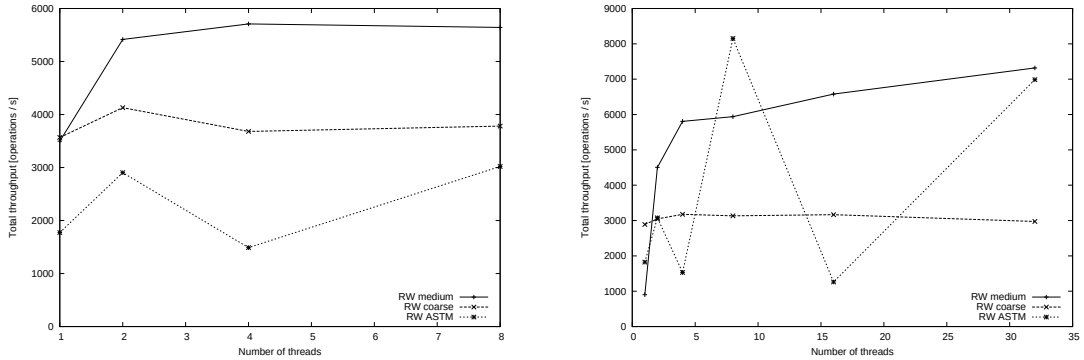
To test our benchmark, we have implemented a version of STMBench7 synchronized using a variant of ASTM—an STM framework available as a Java library. The tests were performed on two machines: a 2-cpu Xeon and an 8-cpu Sun V40z, using JDK 1.6 (beta) and the Polka contention manager included in ASTM.

Our STM-based implementation is a straightforward, and so not necessarily optimal, approach to the problem. We tried to look from the perspective of an average programmer who has chosen STM because it is advertised as being almost as easy to use as coarse-grained locking. Thus, we

Read-dominated workload



Read-write workload



Write-dominated workload

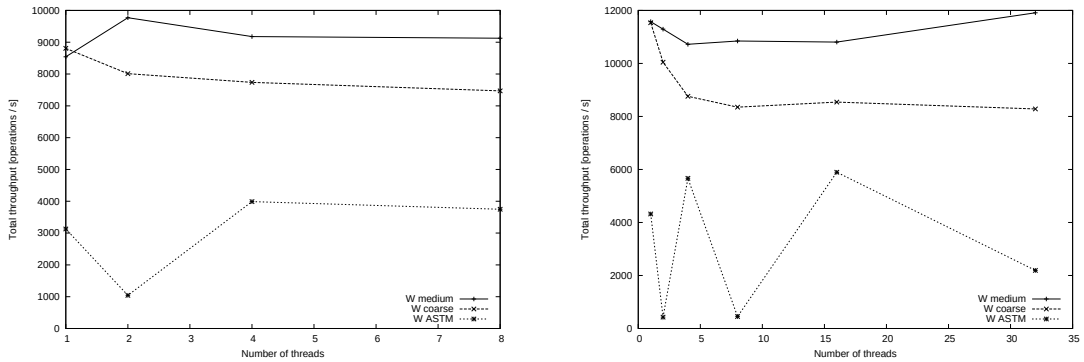


Figure 6: Comparison of ASTM-based synchronization with coarse- and medium-grained locking strategies included in the STMBench7 suite with all long operations disabled. Left: 2-cpu Xeon, right: 8-cpu Sun V40z.

Threads	Workload type					
	Read-dominated		Read-write		Write-dominated	
	Lock	ASTM	Lock	ASTM	Lock	ASTM
1	2396	1.1	1361	1.60	813	6.6
2	2982	1.6	1396	1.5	814	8.9
4	2976	2.1	1430	2.3	779	2.1
8	2876	0.7	1343	0.7	788	7.6

Table 3: Comparison of total throughput (operations per second) for coarse-grained locking and ASTM, with long traversals disabled

made each non-immutable object in the data structure transactional[2] and converted each operation to a single, flat transaction. Of course, the performance results we provide could be used for further improvements of the STM implementation.

Our simple ASTM-based implementation performs very poorly when long traversals are enabled—a single execution of traversal T1, for example, could last as much as half an hour (with a single thread, on the 2-cpu machine; as compared to about 1.5 s for locking). With long traversals disabled, we got the throughput results presented in Table 3.

The cause of the poor performance of the ASTM-based STMBench7 are two kinds of operations: the ones that acquire a large number of objects in read mode and the ones that perform updates on very large objects (like the manual). The reason for that are two elements of the ASTM design: *invisible reads* and *logging granularity*, i.e., object-level granularity of logging changes made by running transactions. More precisely, when a transaction acquires an object in a read-only mode, it adds the object to its private list. The list is not visible to other transactions. Therefore, an object acquired for reading can be subsequently acquired for writing by another transaction. This means that, in a general approach, which is also used in ASTM, a transaction has to validate its private list every time the transaction acquires an object for reading. Thus, the cost of validation for every transaction is $O(k^2)$, where k is the number of objects the transaction acquired for reading. This explains the problem with long traversals, some of which, in our ASTM-based implementation of STMBench7, have to acquire more than 50 millions of objects.

ASTM performs logging of changes made by transactions by copying the objects that were acquired for writing. Therefore, even if only a single attribute of an object is changed, a copy of the whole object has to be made. This clearly poses a problem, because the manual and each index are represented by single objects. As our ASTM-based implementation does not split large objects into smaller parts, the performance of operations that updates these objects is significantly limited.

A solution would be to group small objects and split the large ones. For example, one could make composite parts contain, logically, all their atomic parts. Then, only composite parts would be transactional and thus the cost of read-only traversals would be significantly lowered. However, composite parts would then become big objects, updates to which would be quite costly. One can also split the manual into a number of chunks, each being a separate transactional object. The indexes could be implemented manually, using, for example, B-trees, with each node synchronized separately—this would make them highly scalable data structures. Nevertheless, one can easily see that if such amount of changes is necessary to use STM in an optimal way, the software engineering advantages of STMs become less visible. In this sense, STMBench7 requires a tough job from STMs and as such it becomes even more interesting.

To check that our suspicions are correct, we disabled all operations that acquire too many objects in read mode or modify either the large index of atomic parts or the manual. The resulting data structure, with remaining set of supported operations, resembles applications that are based on short queries over partially static, tree-based data structure. This come close to the synthetic benchmarks that have been used for evaluating STMs so far (e.g., [7, 9, 5]).

We repeated all the experiments with the so-modified STMBench7. The results, presented in Figure 6, confirm that ASTM performs very well in some, specific scenarios. Namely, for read-dominated workload ASTM-based synchronization is as scalable as medium-grained locking (see the plot for the 8-cpu machine) and outperforms

[2]Transactional objects are shared objects access to which is controlled by an STM.

coarse-grained locking if enough processors and threads are available. This should not be surprising. The ASTM-based implementation, however, seems to behave in a quite instable way when the ratio of update operations is larger. Unfortunately, we do not understand the cause of this behavior at the time of the submission.

6 Summary

This paper presents a first step towards a benchmark for evaluating software transactional memory implementations. STMBench7 has the following desirable properties:

- Its data structure and workload are realistic and correspond to an important class of applications (e.g., CAD, CAM or CASE software).
- The data structure is dynamic thus exercising the aspects of STMs related to memory allocation.
- Tests that are known to be problematic for STMs such as long traversals and complex objects are included. In a sense, STMBench7 can be viewed as a "crash test" for software transactional memory.
- The set of input parameters is small and has intuitive semantics, which make the benchmark easy to use.
- The output is very detailed, allowing for in-depth analysis of performance bottlenecks.
- It provides two lock-based synchronization mechanisms that can be used to set a baseline for comparison.

STMBench7 is open source and can be downloaded from the authors' web site[3].

Clearly, STMBench7 is in a preliminary stage and more experiments will help evolve our benchmark. For instance, adding a fine-grained, highly-optimized locking strategy would help define the "ultimate" baseline test of STMs. Also, more workloads need to be explored.

Acknowledgments

We are very grateful to Mark Moir for his comments and suggestions and to Doug Lea for his help with the experiments.

References

[1] The OO7 benchmark. `ftp://ftp.cs.wisc.edu/oo7`.

[2] RSTM—the Rochester software transactional memory runtime. `http://www.cs.rochester.edu/research/synchronization/rstm`.

[3] STMBench7 home page. `http://lpd.epfl.ch/kapalka/stmbench7.php`.

[4] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):12–21, 1993.

[5] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, 2006.

[6] M. Herlihy. SXM software transactional memory package for C#. `http://www.cs.brown.edu/~mph`.

[7] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, 2003.

[8] 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*, pages 289–300. May 1993.

[9] V. J. Maranthe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)*, pages 354–368, 2005.

[10] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of software transactional memory. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (Transact'06)*, 2006.

[11] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, 2006.

[12] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213. Aug 1995.

[13] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, 2006.

Appendix

A STMBench7 User's Guide

A.1 Running the Benchmark

To run the benchmark you will need:

- JRE 1.5 or later,
- AspectJ run-time libraries (`aspectjrt.jar`, tested with version 1.5).

The easiest way to run STMBench7 is to use the `run` script included in the source code archive. It accepts the same command-line parameters as the benchmark, i.e.:

- `-t numThreads`
  set the number of threads (default: 1),
- `-l length`
  set the length of the benchmark (in seconds),
- `-w r|rw|w`
  set the workload type (`r` = read-dominated, `rw` = read-write, `w` = write-dominated; default: read-dominated),
- `-g coarse|medium`
  set the lock granularity (will fail if the benchmark compiled without the built-in lock strategies),
- `--no-traversals`
  disable long traversals,
- `--no-sms`
  disable structure modification operations,
- `--ttc-histograms`
  print TTC (latency) histograms to `stdout`.

The output of the benchmark, printed to `stdout`, is divided into the following sections:

1. *Benchmark parameters*—a list of input parameters for a given benchmark run.
2. *TTC histograms* (only if `--ttc-histograms` command-line option used)—a list of latency histograms. For each operation named X a single line is printed: `TTC histogram for` X`:` followed by a space-delimited list of pairs $ttc, count$, which say that a $count$ number of operations X have completed in time from range $[ttc, ttc - 1]$ (in milliseconds).
3. *Detailed results*—for each operation named X the following data is displayed: (1) the number of times X has been successfully completed (by any thread), (2) the maximum time it took to successfully complete X (in milliseconds), and (3) the number of times X started but failed before completion.
4. *Sample errors*—for each operation type (category) named T the following data is output: (1) the ratio C_T for operations of type T computed from the input parameters, (2) the ratio R_T of the number of successful operations of type T to the number of all successful operations in the current benchmark run, (3) computed error: $E_T = |C_T - R_T|$, (4) the ratio A_T of the number of successful and failed operations of type T to the number of all successful operations, and (5) computed error: $F_T = |A_T - R_T|$.
5. *Summary results*—contains the following information:
   - For each operation type named T the following data is displayed: (1) the number times operations of type T have been successfully completed, (2) the maximum time it took to successfully complete any of the operations of type T, (3) the number of times operations of type T started but failed before completion, and (4) the number of times operations of type T started (and completed or failed).
   - The total sample errors E and F computed as a sum of all E_T, or F_T, respectively, numbers for every type T.
   - The total throughput given with two numbers, both in operations per second: (1) the total number of operations that have been successfully completed divided by the elapsed time, and (2) the total number of operations that started (and completed or failed) divided by the elapsed time.
   - The elapsed time, in seconds.

All other messages are printed to `stderr`. These are either error messages, or information not necessary for interpreting the benchmark results.

A.2 Compiling the Benchmark

To compile the benchmark code you will need:

- JDK 1.5 or later,
- AspectJ (tested with version 1.5), if locking strategies are to be merged with the benchmark core code,
- GNU `make` (not strictly necessary, but makes the build process much easier).

To unpack and build STMBench7 with locking strategies compiled-in:

1. `tar -xvzf`
2. `stmbench7-`*`version`*`.tgz`
3. `cd stmbench7-`*`version`*
4. `make`

To unpack and build STMBench7 without any locking strategies compiled-in:

1. `tar -xvzf`
2. `stmbench7-`*`version`*`.tgz`
3. `cd stmbench7-`*`version`*
4. `make build-without-locking`

Note that this version will not work by default. It should be first synchronized using an STM, a custom locking scheme or another mechanism.

To create a JAR file, type: `make jar`.

B STMBench7 Specification

B.1 The Data Structure

See Section 2 and the specification of OO7 [1]. What is important from the synchronization perspective is that only the module and connection objects are immutable. The others, including indexes, sets and bags, can be updated by STMBench7 operations.

The specification of OO7 does not say precisely, how some of the attributes of the created objects should be initialized. Please, see the STMBench7 source code to see how we do it. This is mostly consistent with the way objects are initialized in the source code of OO7, with exceptions being the result of some "cosmetic" changes.

B.2 The Operations

In this section we describe all 45 operations of STMBench7. As already mentioned, each operation should be executed atomically, i.e., the results of all updates made by an operation should become visible to others (logically) at a single point in time.

B.2.1 Long Traversals

All long traversals originate from traversals and queries of OO7. The original naming has been preserved. Long traversals can never fail.

1. T1: traverse the whole structure depth-first, starting from the module and the root complex assembly. That is, for each complex assembly traverse all its sub-assemblies, for each base assembly traverse the descendant composite parts, for each composite part traverse, depth-first, the graph of its atomic parts, and perform a read-only operation on each atomic part. Return the number of atomic parts visited.
2. T2a: the same as T1, except that an update operation on non-indexed attributes (x and y) is performed on each root atomic part.
3. T2b: the same as T1, except that an update operation on non-indexed attributes is performed on each atomic part.
4. T2c: the same as T2b, except that each update on an atomic part is performed 4 times, one-by-one.
5. T3a: the same as T1, except that an update operation on an indexed attribute (*buildDate*) is performed on each root atomic part.
6. T3b: the same as T1, except that an update operation on an indexed attribute is performed on each atomic part.
7. T3c: the same as T3b, except that each update on an atomic part is performed 4 times, one-by-one.
8. T4: traverse the structure depth-first, from the module and the root complex assembly down to all the document objects. That is, for each complex assembly traverse all its sub-assemblies, for each base assembly traverse its descendant composite parts, and for each composite part perform a read-only operation

(search for character "I" in the text of the document) on its descendant document object. Return the total number of "I" characters in all documents.

9. T5: the same as T4, except that an update operation is performed on each document object (replace "I am" by "This is", or vice versa, in the text of the document). Return the number of replaced sub-strings in all documents.

10. T6: the same as T1, except that only the root atomic part is visited (i.e., there is no depth-first search on each graph of atomic parts).

11. Q6: find all complex assemblies that are ascendants of some base assembly such that *buildDate* of the base assembly is lower than *buildDate* of at least one of its descendant composite parts. That is, for each complex assembly traverse its sub-assemblies and for each base assembly iterate through its descendant composite parts until a one with larger *buildDate* is found. Perform a read-only operation on every assembly that matches the query. Return the number of matched assemblies.

12. Q7: iterate through all atomic parts, using the atomic part ID index, and perform a read-only operation on each of them. Return the number of atomic parts visited.

B.2.2 Short Traversals

Read-only short traversals:

1. ST1: traverse the structure top-down, from the module to an atomic part, via a random path. That is, for each complex assembly traverse its random sub-assembly, for each base assembly traverse its random descendant composite part (if any) and for every composite part perform a read-only operation on its random descendant atomic part (traversing the graph of atomic parts is not necessary for each composite part contains a set of pointers to its descendant atomic parts). Return the sum of the attributes x and y of the visited atomic part. The traversal fails if a base assembly with no descendant composite parts is visited.

2. ST2: traverse the structure top-down, from the module to a document, via a random path (similarly to ST1). Return the number of "I" characters in the text of the visited document. The traversal fails if a base assembly with no descendant composite parts is visited.

3. ST3 (T7 in OO7): traverse the structure bottom-up, from a randomly chosen atomic part (using the atomic part ID index) to the root complex assembly. That is, choose a random atomic part ID, find the corresponding atomic part (fail if not found), go to its parent composite part, and for each ascendant base assembly (fail if none) traverse its ascendant complex assemblies up to the root one. Visit, however, each complex assembly at most once. Perform a read-only operation on each visited complex assembly and return the number of complex assemblies visited.

4. ST4 (Q4 in OO7): generate 100 random document titles and find the corresponding document objects using the document title index. Perform a read-only operation on each base assembly ascendant (via a composite part) of at least one of the found documents. Return the number of base assemblies visited.

5. ST5 (Q5 in OO7): find all base assemblies such that their *buildDate* is lower than *buildDate* of some of their descendant composite parts. Perform a read-only operation on each found base assembly and return their number. This short traversal does not traverse the complex assemblies—it iterates over the index of base assembly IDs.

6. ST9: the same as ST1 except that all atomic parts descendant of a given composite part are visited—a depth-first search on their graph is performed. Returns the number of atomic parts visited.

Non-read-only equivalents of some of the short traversals ST1–ST9:

1. ST6: the same as ST1, but performs an update operation on non-indexed attributes of the visited atomic part.

2. ST7: the same as ST2, but performs an update operation on the (non-indexed) text of the visited document (replace "I am" by "This is", or vice versa, in the text of the document). Returns the number of sub-strings replaced.

3. ST8: the same as ST3, but updates each visited assembly (the non-indexed *buildDate* attribute).

4. ST10: the same as ST9, but performs an update operation on non-indexed attributes of all the visited atomic parts.

B.2.3 Short Operations

Read-only short operations:

1. OP1 (Q1 in OO7): choose 10 random atomic parts, using the atomic part ID index, and perform a read-only operation on each of them. Return the number of atomic parts that were processed (this may be lower than 10 for some index lookups may fail as IDs are chosen randomly).

2. OP2 (Q2 in OO7): find all atomic parts that have *buildDate* in range $[1990, 1999]$ using the atomic part build date index. Perform a read-only operation on each atomic part found. Return the number of processed atomic parts.

3. OP3 (Q3 in OO7): the same as OP2, but the range is $[1900, 1999]$.

4. OP4 (T8 in OO7): count the number of occurrences of character "I" in the text of the manual. Return the computed number.

5. OP5 (T9 in OO7): check if the first and the last characters of the text of the manual are the same. Return 1 if true, 0 if not.

6. OP6: choose a random complex assembly, using the complex assembly ID index, and perform a read-only operation on all its sibling complex assemblies. Return the number of complex assemblies processed. Fail if the randomly chosen ID does not correspond to any existing complex assembly.

7. OP7: choose a random base assembly, using the base assembly ID index, and perform a read-only operation on all its sibling base assemblies. Return the number of base assemblies processed. Fail if the randomly chosen ID does not correspond to any existing base assembly.

8. OP8: choose a random base assembly, using the base assembly ID index, and perform a read-only operation on all its descendant composite parts (if any). Return the number of processed composite parts. Fail if the randomly chosen ID does not correspond to any existing base assembly.

Non-read-only equivalents of some of the operations OP1–OP8:

1. OP9: the same as OP1, except that it performs an update operation on non-indexed attributes of every visited atomic part.

2. OP10: the same as OP2, except that it performs an update operation on non-indexed attributes of every visited atomic part.

3. OP11: replaces all the occurrences of character "I" with character "i", or vice versa, in the text of the manual. Returns the number of changes made.

4. OP12: the same as OP6, except that an update operation is performed on each visited complex assembly.

5. OP13: the same as OP7, except that an update operation is performed on each visited base assembly.

6. OP14: the same as OP8, except that an update operation is performed on each visited composite part.

7. OP15: the same as OP1, except that it performs an update operation on the indexed *buildDate* attribute of every visited atomic part.

B.2.4 Structure Modification Operations

1. SM1: create a composite part, with its corresponding document and a graph of atomic parts, and add it to the design library (without linking to any base assembly). Fail if the maximum number of composite parts has been reached.

2. SM2: delete a composite part with a randomly chosen ID, together with its corresponding document and the graph of descendant atomic parts. Fail if the lookup operation on the composite part ID index fails.

3. SM3: create a link between a base assembly and a composite part with randomly chosen IDs. Fail if any of the index lookup operations (on the base assembly ID index or the composite part ID index) fails.

4. SM4: chose a random base assembly ID and find the corresponding base assembly (fail if the index lookup operation fails). Then, delete

a randomly chosen link between the base assembly and some composite part.

5. SM5: add a base assembly in the following way. First, chose a random base assembly ID and find the corresponding base assembly B (fail if the index lookup operation fails). Then, create a base assembly B' as a sibling of the base assembly B. Fail if the maximum number of base assemblies has been reached.

6. SM6: delete a base assembly with a randomly chosen ID. Fail if the lookup operation on the base assembly ID index fails or if the chosen base assembly is the only descendant of its parent complex assembly.

7. SM7: add an assembly sub-tree under a randomly chosen complex assembly. First, choose a random complex assembly ID and find the corresponding complex assembly C (fail if the index lookup operation fails). Then, add an assembly sub-tree of degree 3 and height $k-1$ under node C, where k is the level at which C is placed in the assembly tree. The added subtree must have base assemblies at its level 1 and complex assemblies at all its higher levels, if any. (Note that we count levels bottom-up, so that base assemblies are on the level 1 and the root complex assembly is on the level 7). Fail if at any point the maximum number of complex or base assemblies has been reached.

8. SM8: choose a random complex assembly ID and find the corresponding complex assembly C (fail if the index lookup operation fails). Delete the whole subtree of assemblies (base and complex) descendant from, and including, C. Fail, however, if C is the root complex assembly or C is the only descendant of its parent complex assembly.