Collecting Transactional Garbage

Fadi Meawad, Ryan Macnak, Jan Vitek
Purdue University

Abstract

Transactional memory holds some promise to improve the practice of concurrent programming, but achieving acceptable performance remains an issue for large-scale adoption of the technology. Implementations of software transactional memory increase the cost of many frequently executed operations and have subtle interactions with the run-time system of the host language. We have observed that implementations significantly increase the load placed on the host's memory subsystem by increasing the allocation rate and altering the lifetime of allocated data. In managed languages this translates to added pressure on the garbage collector, which must efficiently reclaim the objects used by the STM to implement transactions. This paper presents experimental data on several STM libraries running on top of Java, and shows that memory pressure significantly impacts their performance. We show similar results for a C# STM tightly integrated with its host virtual machine. Finally, we present performance results that demonstrate that adding support for transactions in the memory subsystem leads to significantly better throughput.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming–parallel programming; D.3.4 [*Processors*]: Compilers; D.1.5 [*Programming Techniques*]: Object-oriented Programming; H.2.4 [*Systems*]: Transaction Processing

General Terms Compilers, Memory, Optimizations

Keywords software transactional memory (STM), garbage collection (GC).

1. Introduction

The case for transactional memory has been made many times. As chip manufactures increase the number of cores in commodity hardware, future performance gains must look to exploiting parallelism. In shared-memory systems, lock-based concurrency control is difficult to use in a reliably correct manner. Transactional memory offers an alternative that can potentially exploit more parallelism than coarse-grained locking, and be simpler to use than fine-grained locking. Transactional code is easier to reason about and avoids problems such as deadlock and priority inversion. While hardware support is likely needed to outperform traditional concurrency control mechanisms, most transactional memory systems are implemented in software so as to facilitate adoption. In many cases, Software Transactional Memory (STM) is implemented as a library on top of some host language. Only a handful of systems integrate transactional support into the language implementation, with changes that crosscut the compiler and the language's run-time system. Even when hardware support is present, it is still the case that long-lived transactions are implemented in software.

From a distance, an STM implementation must take care of two things: first it must detect conflicts between concurrent transactions, and second it must provide support for aborting a transaction and undoing its memory effects. To do this it must maintain auxiliary data structures that record operations performed by transactions. These data structures remember which objects or memory locations were accessed and record enough information to recover their original state. There are several challenges for STM implementations. The STM must deal with features of the host language that cannot be undone such as interactions with external devices, it must attempt to keep the overhead on frequently executed operations such as reads and writes to a minimum, and it must deal with language features that hinder optimizations, such as reflection.

As many STM systems are built on top of managed languages, such as Java, C#, Haskell and ML, it is natural for implementers to leverage parts of the run-time system of the host language. In particular, the memory subsystem and its garbage collection algorithm are often used to manage the allocation and reclamation of the auxiliary data structures used by the STM. This choice entails increased pressure on the Garbage Collector (GC). Typically, the STM requires objects to have additional fields, thus increasing the footprint of the system even when transactions are not used. When transactions are used, logs and copies of objects may have to be allocated. Transactions can also extend object lifetime. Otherwise unreachable values may be referred to from a log, and thus kept alive longer than necessary. Lastly aborts make all of the objects created in the transaction unreachable, pushing onto the GC a batch of objects that need to be collected that would never have even been created under a corresponding lock-based implementation.

We first became interested in the interaction between STMs and GC while working with the Bartok STM [4]. Our C# implementation of the STMBench7 [3] concurrency benchmark exhibited surprising pathologies. We saw that when the benchmark ran with a workload containing long-lived transactions ("long traversals") the program spent over 60% of its time in GC. Furthermore as we increased the number of active threads the ratio quickly shot up to close to a 100%. Interestingly, without long traversals, the ratio is markedly smaller though still shockingly high for a benchmark that

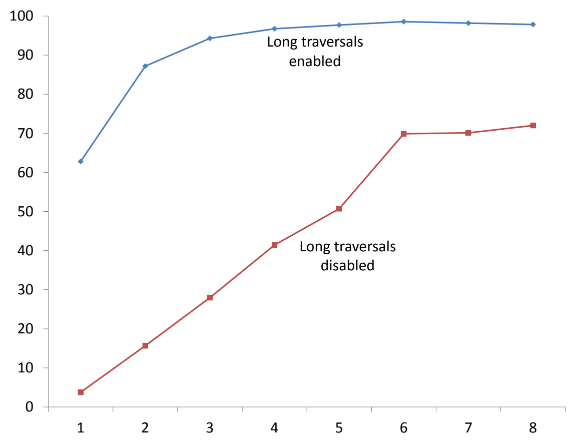

Figure 1. Time in GC. Running the C# STMBench7 on Bartok. The x-axis gives the number of threads, and the y-axis the percentage of the total time spent doing GC.

does little allocation. Fig. 1 shows these results and a description of the Bartok environment is given in Sec. 4.1.

We wondered if we were observing behavior particular to Bartok and its implementation or if this was a symptom of a more general problem? We started by surveying STM implementations for garbage-collected languages looking for any integration with the GC or mention of design decisions based on GC considerations. We found little integration. Most systems are library-based and do not have the option of modifying the host run-time. The C# systems include Bartok [4], LibCMT[1], Microsoft's SXM, and XSTM [10]. Bartok is the outlier with its STM system integrated into the compiler and the memory subsystem tasked with log compaction before GC and use of weak references. Java systems include ASTM [9], AtomJava [7], Deuce [8], DSTM [5], DSTM2 [6], Multiverse[2], and the Intel STM [1]. Most of them either translate source code or recompile code ahead of time. Only the Intel STM uses an altered JVM. Part of the value of Java is the near-universal deployment of the JVM, which is lost with a modified VM. None of the implementations use weak references. This is probably because they are implemented by wrapper objects, and would add overheads. Deuce mentions that they "limit as much as possible the stress on the garbage collector, by using object pools when keeping track of accessed fields." DSTM mentions remembering to null-out fields to make objects eligible for collection.

The contribution of this paper lies in our evaluation of the interaction between transactions and GC in three existing STMs—Deuce, Multiverse and Bartok. We have found the transaction system places significant additional pressure on the GC. We describe an optimization that allocates the transactional data on a dedicated heap, separate from the normal heap, and its implementation in Bartok. We measured the effect of this optimization, and found it helped the most for benchmarks with larger transaction sizes.

2. Benchmarks

We evaluate STM implementations on the following four benchmarks. All benchmarks used in this paper are freely available from the project's web page[3] along with links to original sources.

2.1 GCBench

GCBench is a new micro-benchmark which creates a linked list, where each element of the list is itself a linked list of wrapped doubles. The benchmark traverses the list in a transaction with, at every node, a probability of (a) updating the node, (b) allocating an unreachable object, (c) allocating an object that will become unreachable after commit, (d) allocating an object that will survive the transaction, and (e) making an object unreachable. These parameters, combined with the linked list size and the number of threads, allows controlling the size of read and update logs, allocated and discarded objects within each transaction. We use the following values: 5% updates, 15% immediately unreachable, 65% unreachable at commit, 5% insert, 5% delete. Each thread operates on its own private data structure, so there is no contention and there will be no GC pressure due to aborted transactions.

2.2 STMBench7

STMBench7 [3] is a benchmark over a data structure composed of trees, graphs and indices intended to be suggestive of CAD/CAM workloads. It has a relatively high memory overhead, with the initial data structure measuring on the order of 500MB. STMBench7 allows the choice of a read-dominated, read-write or write-dominated workload, as well as whether to enable long traversals (operations that touch most of the data in one transaction). Operations generally do not allocate with the exception of a hash table to maintain a set. The exception would be the structural modification operations, which we did not enable. The original implementation of the benchmark is in Java; we ported it to C# for these experiments.

2.3 LeeTM

The LeeTM [11] benchmark performs automatic circuit routing using Lee's algorithm. Pairs of points on a grid that could represent transistors on an integrated circuit, logic elements on an FPGA, or packages of ICs on a circuit board, are connected with non-intersecting paths. The task has a large degree of potential parallelism. It is non-trivial to come up with a locking scheme that is more fine-grain than simply locking the whole board for routing each path, yet most routes typically will not conflict. This is the kind of scenario where STM should have the greatest advantage. Various workloads are available by routing different circuits, ranging from simple short parallel paths to a memory circuit and a motherboard. We used versions of LeeTM that do and do not have allocation in transactions.

2.4 WormBench

WormBench [12] is a configurable benchmark that measures the performance and robustness of STM systems. It is implemented in C#, and was designed originally to run with the Bartok STM. The main idea is to have a "worm" with a triangular-shaped head and a line-shaped tail that lives in a matrix with other worms. The worms have 15 different operations, such as move forward or turn right, that they perform atomically so they do not end up occupying the same spaces as other worms. There is no allocation within transactions in WormBench. As of this writing WormBench is only available for C#.

3. Evaluating the impact of GC on STMs

We now describe the results of our evaluation of the impact of GC on the performance of STMs. Unless noted otherwise, all benchmarking was done on an 8-core, 1.60GHz Intel Xeon E5310 with 8GB of RAM and Physical Address Extension enabled, running Windows Server 2003 SP2. Our Java virtual machine was HotSpot 1.6.0_21, and unless otherwise specified we ran with a 1.5GB heap size and the default collector (see Sec. 3.4). We also used an Azul Vega 3 3310B, with two 54-core processors and 48GB of RAM. The benchmark ran on top of the Azul Virtual Machine with the Concurrent Pauseless GC [2].

3.1 GCBench

We ran GCBench with two library-based Java STMs (Deuce and Multiverse) and the Bartok C# STM. Since the different threads in this benchmark operate on distinct data, we use an unsynchronized version of the benchmark to give us a lower bound on execution time. Any GC activity solely due to the application will show up in the unsynchronized version. Any difference in GC activity can be attributed to the STM. Table 1 shows the results for the Java. Already with one thread Deuce and Multiverse are 10.2x slower than raw Java. Due to what we assume to be an implementation defect, Deuce deadlocks after three threads. Multiverse with 8 threads performs 16.7x slower than raw Java. At that point Multiverse spends 77% of its time in GC. Unsynchronized Java spends 81% in GC. The difference between the raw Java GC time (39 secs with 8 threads) and the Multiverse GC time (588 secs) comes solely from the allocations performed by the STM infrastructure. While Deuce does not scale as well, the same trend of increased GC work can be observed up to three threads.

[1] `http://sourceforge.net/projects/libcmt`

[2] `http://multiverse.codehaus.org` [3] `http://bit.ly/fMueU6`

For the C# version we report the running time of the benchmark for different numbers of threads (1 to 8) and different problem sizes (100 to 800). Fig 2 shows those results. Table 2 is the percentage of time spent in GC. For any given size, increasing threads will increase the application's allocation rate and thus create more work for the GC (and slow down the whole system). The data illustrates this and shows the overheads of the STM implementation.

T	Deuce		Multiverse		Unsynced	
	GC	Total	GC	Total	GC	Total
1	31.9	99.5	18.8	98.5	4.7	9.7
2	70.6	223.6	46.8	145.9	10.0	17.5
3	121.5	789.3	69.7	183.9	13.6	20.2
4			87.3	208.1	18.7	30.5
5			118.4	244.8	24.3	32.5
6			181.7	317.6	28.5	40.9
7			298.4	449.8	37.1	44.1
8			588.4	769.2	39.4	46.9

Table 1. Java GCBench. Execution time in seconds. Size 800, Java version, with 1.5GB heap. Deuce deadlocks with more than 3 threads.

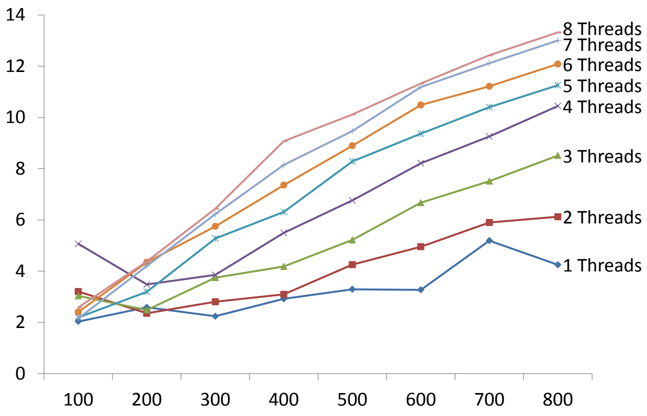

Figure 2. C# GCBench. The x-axis is list problem size, and the y-axis is the slowdown between STM and unsynchronized C# on Bartok. Each line represents a different number of threads.

T	100	200	300	400	500	600	700	800
1	0%	9%	23%	36%	48%	54%	61%	67%
2	12%	35%	56%	69%	74%	82%	85%	88%
3	25%	53%	72%	79%	87%	91%	93%	95%
4	39%	66%	81%	88%	92%	95%	96%	97%
5	40%	75%	87%	92%	95%	96%	97%	98%
6	49%	81%	90%	95%	97%	98%	98%	99%
7	60%	82%	93%	96%	97%	98%	99%	99%
8	64%	85%	94%	97%	98%	99%	99%	99%

Table 2. C# GCBench. Percentage of total time spent in GC for Bartok with different problem sizes.

3.2 STMBench7

As mentioned in the introduction our experience with STMBench7 on Bartok was the motivation for this work. We found that it scales poorly even on read-dominated (low contention) workloads. With long traversals disabled, throughput is never more than 31% faster than single thread performance even up to 8 threads; with long traversals enabled, it never goes faster than at a single thread (Fig. 7). As the number of threads increases, the fraction of time spent in GC increases, approaching 70% and 98% respectively (Fig. 1). An example of a long-traversal operation is *Traversal1*, which performs a depth-first search through the data structure and builds 500 sets of 100000 elements total in the process. Sets are implemented by hash tables, so some allocation occurs whenever they need to resize. But more importantly, Bartok must keep log entries for the reads of the 1 million+ elements of the data structure. Because the operation takes longer than the time between GCs, the logs survive collection and the heap grows. Growing heaps trigger more collections. Larger heaps take more time to collect.

T	Deuce				Multiverse			
	Short		Long		Short		Long	
	GC	Total	GC	Total	GC	Total	GC	Total
1	155.4	601.2	69.9	620.3	0.0	5.0	0.1	15.0
2	89.9	602.3	80.5	663.1	0.1	3.4	0.1	9.0
3	41.0	604.7	168.5	685.8	0.0	3.3	0.3	7.0
4	27.9	629.7	290.3	745.1	0.3	3.0	0.4	6.0
5	29.1	622.7			0.0	3.0	0.1	5.0
6	34.9	635.2			0.0	3.0	0.2	4.0
7	22.8	809.3			0.2	3.0	0.2	4.0
8	22.5	643.9			0.2	3.0	0.3	3.0

Table 3. LeeTM. GC and Total execution time in seconds for short and long traversals using Deuce and Multiverse.

3.3 LeeTM

Table 3 shows LeeTM performance and GC overhead for Deuce and Multiverse. Deuce short does not exhibit a large GC overhead; the overall performance is 5x slower than Bartok (Fig. 9). Multiverse on this specific benchmark is able to reduce the GC overhead as well as minimizing the overall time. Its performance is comparable to the Opt versions of LeeTM for Bartok, presumably because it does a better job of handling transaction-local objects (discussed further in Sec 5.3).

3.4 Choice of Collector

We wanted to ensure that our results were not due to a peculiarity of the garbage collection algorithm. HotSpot offers the choice of three different algorithms: serial (default), parallel, and concurrent mark-sweep. All of the collectors are generational. Parallel is a stop-the-world collector that uses multiple threads. Concurrent mark-sweep uses a separate thread to do GC work concurrently with application work, but it does include a STW phase for major collections. We tested them all on GCBench with a fixed max heap size of 1.5GB and found that the serial collector performed best with higher numbers of threads for Multiverse, while for the unsynchronized version it was the parallel collector that slightly outperformed the others (Fig. 3). The choice of collector did not change our conclusions.

3.5 Scalability with Azul

To examine how the overhead scales beyond 8 cores, we switched to an Azul machine with 108 cores. Fig. 4 shows the results of a run of GCBench for data size of 800 and a heap size of 20GB (out of the 29GB available for applications). For Multiverse, with each increment of 5 threads, the benchmark slows down by an extra 10% except at the very end, where the last data point is 61% slower than the one before. At that data point, the benchmark is utilizing all of the heap (Fig. 5), triggering more collections. (Azul grants extra memory from the grant pool for up to 1G). Despite the presence of a concurrent GC, threads will block waiting for more memory to be made available. The total number of cores used is greater

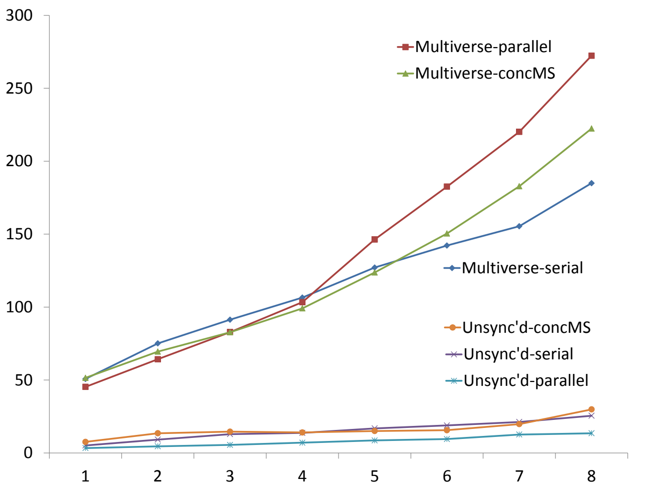

Figure 3. Collectors. GCBench size 600 with Multiverse and unsynchronized Java. The y-axis is execution time in seconds and the x-axis is the number of threads. Lines show the garbage collection algorithms available in Hotspot (Serial/Parallel/ConcMS). There is about 90MB of live data per thread in Multiverse and 20MB for unsynchronized Java.

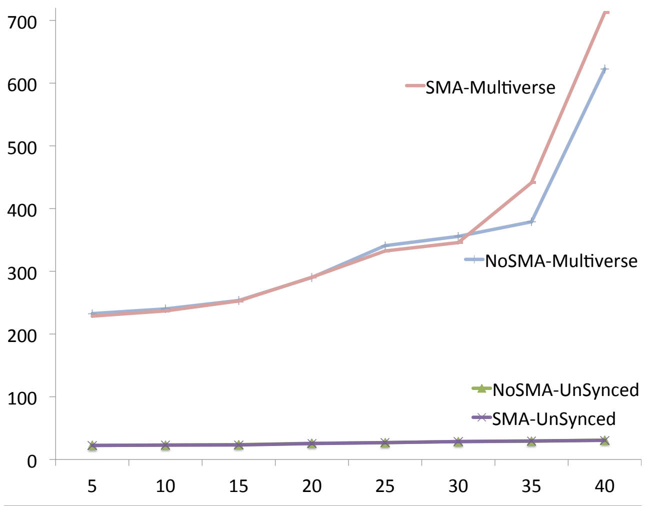

Figure 4. Java GCBench on Azul. Unsynchronized and Multiverse versions of GCBench with size 800. Y-axis is time in seconds, x-axis is number of threads. SMA means Speculative Multi-address Atomicity is enabled.

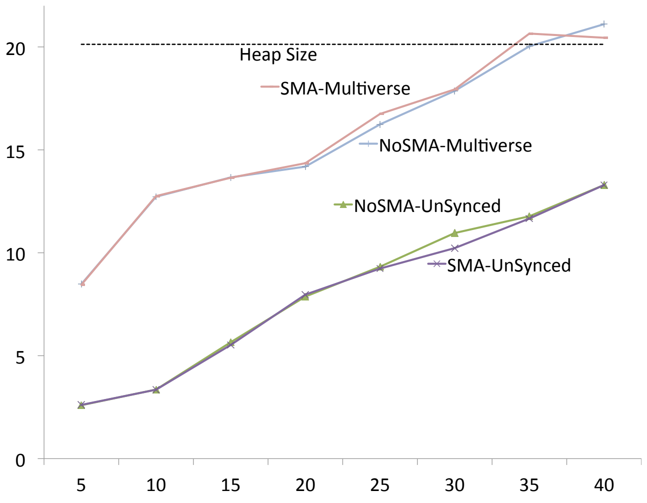

Figure 5. Java GCBench on Azul. Peak memory usage. Y-axis is peak memory in GB, x-axis is number of threads.

than the number of threads because additional cores are used by the GC, but it is still less than the number of hardware cores. The benchmark is limited by the available memory. The distance between the performance of Multiverse and raw Java is striking: the culprit is the overheads due to the transactional operations (reads and writes).

The Azul appliance has a form of hardware transactional memory called Speculative Multi-address Atomicity (SMA). SMA kicks in whenever a Java `synchronized` statement is executed and speculates it in a transaction. We did not set out to evaluate the benefits of SMA in this work, but we report, for completeness, the results with SMA enabled and with SMA turned off. For the unsynchronized version of GCBench there is, as expected, no difference. For the version of GCBench running with Multiverse, SMA slightly decreases performance for higher thread counts.

3.6 Discussion

The overheads we have observed point to pathologies that make the STM systems we have tried all but useless for the workloads under consideration. By varying the implementation strategy (library vs. integrated in the VM), the choice of GC algorithm (serial, parallel, concurrent mark-sweep and pauseless), the numbers of threads (1 to 40) and the workloads, we believe that we have shown conclusively that memory pressure is a significant problem for performance of STM implementations.

4. A Transaction-aware GC

This section describes how we modified the memory subsystem of one particular infrastructure, Bartok, to make it transaction aware.

4.1 Bartok

The Bartok STM [4] is an object-based STM integrated into the Bartok research compiler. The integration in the compiler allows direct access to objects' fields rather than using shadow copies of objects as many library-based implementations do. It uses a decomposed STM interface, which combined with special compiler optimizations allows for the elimination of some logging operations. For example, multiple reads to the same object are recorded by a single read enlistment, and reads to an object are subsumed by writes needing only an update enlistment. Bartok uses three logs for each transaction: read enlistment, update enlistment, and undo values. The logs are allocated in chunks and linked in reverse order (i.e., newest chunk at the beginning). Bartok also uses objects' headers to store which transaction if any has opened the object for update. It is the first STM to allow objects that become unreachable within a transaction to be collected. It performs log compaction in a pre-GC phase, taking advantage of the stopped world to remove duplicate read and update enlistment entries and to remove reads subsumed by updates. It further arranges that log entries other than undo values are treated as weak references, allowing objects to be reclaimed if they are only referenced by read or update enlistment entries. Objects created within a transaction can be collected this way, but any objects that existed prior to the transaction would still be referenced strongly by an undo entry. For our experiments, we use the *Semispace* collector (which is generational, serial and stop-the-world) because it is the best integrated with the STM. However, under high stress workloads, errors still sometimes occur with and without our optimizations.

4.2 Dedicated nurseries

One way to improve performance is to take advantage of STM's impact on object lifetime by having a nursery per transaction. Objects that are allocated within a transaction are not visible to other threads until the transaction commits, so GC traversal of these objects is redundant before commit. Based on our observation of the above mentioned benchmarks, few objects get allocated and discarded voluntarily within a single transaction. Therefore, a simple bump allocator can be used to allocate from this heap. On abort, after restoring undo values, the entire per-transaction nursery can be reclaimed. This can be accomplished easily by resetting the per-transaction nursery heap pointer. On commit, the per-transaction nursery can be promoted to normal nursery. This can be done cheaply by re-labeling its pages as normal nursery pages. This idea can be easily adapted for systems that allow nested transactions. Aborting an inner transaction can be done by resetting the allocation pointer to its value at the beginning of the inner transaction. Commit requires no special handling, since the transaction will truly commit only when the outer-most transaction commits.

The benefits of this optimization depend on the nature of the workload and the other optimizations in the system. Workloads with high contention are likely to see the most improvements as the cost of aborts is lowered and as doomed object will be reclaimed early. This optimization complements the elision of logging operations for transaction-local objects [1]. On the other hand, in some STMs it is preferable to pre-allocate data before the start of the transaction in order to avoid logging the initializers; when this programming idiom is used, this optimization will be less useful.

Finalizable objects pose a concern because discarding a transaction nursery in one step would not allow their finalizers to run. It would not be unreasonable to prohibit finalizable objects in transactions because often they are mostly required for freeing IO resources, and IO is generally not allowed in a transactional scope. Alternatively, one could allocate finalizable objects on the normal heap.

4.3 Dedicated heap

Detailed analysis of the experimental results reveals that 90% of the memory being allocated in Bartok is for transactional logs rather than application objects. Similarly, most of the GC's time is spent on dealing with the STM's data. The designers of the Bartok STM chose to treat all STM data as normal objects, like in a library-based implementation. Short-lived transactions have small logs which are ready for collection quickly. If short transactions are frequent, minor GCs will be triggered often since these logs are filling the nursery. For long transactions, GC will be triggered within the transaction, logs will be promoted, eventually leading to an increase in the number of major collections.

Transactional logs have three important properties: (*a*) they are only reachable from the object which represents the transactional context, (*b*) they are bounded by the transactional scope they serve, (*c*) they are allocated sequentially and are only discarded at transaction end. One possible solution to reduce the cost of managing logs would be to pool and reuse them. But as the log sizes are a function of the number of operations performed in a transaction and thus can vary widely, this would entail some complexity in managing the pool of log objects and making sure that unused logs eventually get reclaimed. Further, this would only reduce the allocation overhead but not remove the need for the GC to trace the pool despite that the pool will not be freed.

We propose a dedicated heap for the logs. With our proposed optimization we would allocate logs from a separate heap and use a dedicate memory manager to allocate and free them. Since the logs are often a major contributor to heap exhaustion, this removes significant pressure from GC. This entails a number of changes that we detail next.

Chunk size. Bartok logs are allocated in chunks containing 512 to 1024 entries, depending on the log type. Large transactions would suffer from smaller log chunk size due to the overhead of frequent allocation/collection. Very small transactions can suffer from allocating a large log chunk. Bartok retains a log chunk per thread for its lifetime (allocated by the first transaction of the thread) to reduce the allocation overhead of small transactions.

To simplify free list management, we use fixed size chunks for all log types. Bartok has three log types: (1) *Read enlistment logs* which store references that were read (entry size is 1 word), (2) *Update enlistment logs* for updated references (entry size is 4 words) and (3) *Undo logs* for original values (2 words). Read logs are smaller but more frequent. Update logs are bigger but less frequent. A log chunk contains a larger number of read enlistment logs than update enlistment logs. The difference in number of entries required by each in a transaction depends on the nature of the transaction's work (mostly-reads vs mostly-writes). Selecting a fixed size for log chunks of all types eliminates unusable fragments in the free list and allows optimizing allocation/free operations.

Allocation. Each thread acquires a lock to the free list in order to allocate or free. Allocating a new chunk is fast (grab the first chunk in the free list). Freeing a log chunk requires more work to add the newly freed chunk to the free list by retrieving the previous free chunk to link to it. Practically this did not cause any measurable overhead if we merge the free space. Since we have fixed size logs, free operations only need to append the newly freed memory to the head of the free list. A better implementation would be to use a wait-free queue, but no contention was observed during our experiments.

An alternative to our approach would be to use a per-thread allocator for the logs. When a transaction ends, its allocator pointer is reset to the beginning of the thread-private log memory. This would avoid the need to fix the logs sizes since no fragmentation can occur. Also no locking is required. However, this design would require acquiring and releasing pages from the system or assigning a fixed share of the memory to each thread.

Nested transactions. We support nested transactions. In case of commit, all logs are kept until the outermost transaction commits as well before freeing any of the logs/log chunks. In case of abort, logs are discarded immediately, and no special handling is required.

Available Memory. A dedicated heap may be seen as reducing the memory available to the application. Especially so if the application uses transactions in an uneven way, for instance starting with long-lived transactions and then switching to short-ones. Periodically, the logs memory manager can release unused pages.

Tracing. Another issue with the dedicated heap is its integration with the GC. Given that Bartok STM performs updates in place, if an object becomes unreachable during a transaction, it has to be preserved until the transaction ends. It can be freed if the transaction commits, but must be restored on an abort. Such objects are only reachable by their log entries. Normally, these objects would be traced by the GC when the log is visited. But since the logs themselves are outside the heap, the GC must be informed of the presence of logs and the objects the logs reference must be traced explicitly.

Stop-the-world. We worked with a stop-the-world garbage collector. Stop-the-world collectors allow the system to synchronize the two heaps operations before and after collection, which will be harder and may require special synchronization in case of a concurrent garbage collector.

4.4 Implementation

As mentioned above, we moved the logs to a separate memory. Log chunks for reads, updates, and undos are allocated from a global free list protected by a spinlock. Since the chunks are large contention for them is minimal. The garbage collector is instructed not to follow pointers into this memory when tracing or copying. During GC, the logs are explicitly visited. The system's bump allocator is now aware of both heaps. According to a runtime flag (which is set before and unset after log allocation), the bump allocator decides which heap to allocate from. The extra check per allocation has a minimal effect (1 extra branch instruction, and 2 memory updates per chunk allocation). Another alternative would be to allocate raw memory from the log heap, and initialize it with the log chunk object information separately, eliminating the test since the decision is made by the code allocating, rather than the allocator. When an outer transaction commits or a transaction aborts, the system does a traversal of the log chunks used (not the logs themselves, only the chunks), and frees each of them individually.

5. Results

We used the testing environment and C# benchmarks described in Sec. 3.

5.1 GCBench

We ran GCBench on the modified STM and compared the running time to the original Bartok STM.[3] Fig. 6 demonstrates the performance improvements of the new implementation. Smaller data sizes (at 100) suffer up to 17% slow down due to the larger log chunks used for small transactions and to the extra check on allocation. As the problem size increases the benefits kick in and peak at a 50% speed up.

5.2 STMBench7

We ran STMBench7's read-dominated workload on Bartok with and without long traversals and compared the performance of the modified STM to that of the original Bartok STM. Fig. 7 shows operations per second. For long transactions the improvements are striking. Fig. 8 gives the relative improvements between the versions of the STM. While, short transaction show improvements of more than 50% after 3 threads, long transactions are around 200% faster.[4] Table 4 gives the log sizes per transaction, and shows that long traversals involve about 5.6x as much logging as short transactions. They benefit more as our modifications prevent these logs from adding to the GC pressure.

	Read	Write	Total
LTD	321816	138991	460807
LTE	2213538	387786	2601325

Table 4. C# STMBench7. Logs by type per average transaction.

5.3 LeeTM

LeeTM allocates transient data structures within each transaction. These objects are transaction-local, but Bartok is not able to detect

[3] At the time of writing, higher thread count triggers a bug in the STM support; we are investigating its source.

[4] The missing observations are due to a software defect in the original Bartok STM.

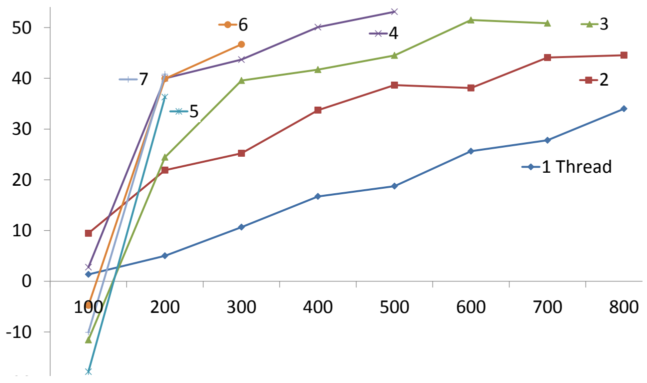

Figure 6. C# GCBench. The y-axis is the percentage improvement in total running time over the original Bartok STM and the x-axis is the problem size. Lines show different thread counts. Missing observations are due to a software defect.

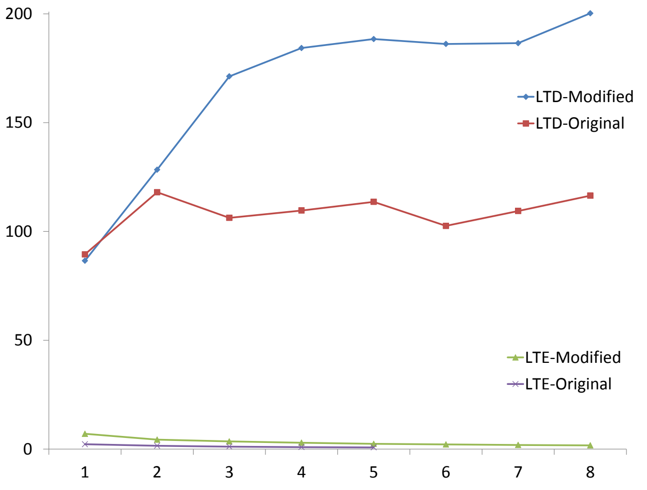

Figure 7. C# STMBench7. The y-axis gives operations per second, and the x-axis gives number of threads. LTD means that long traversals are disabled, and LTE that they enabled.

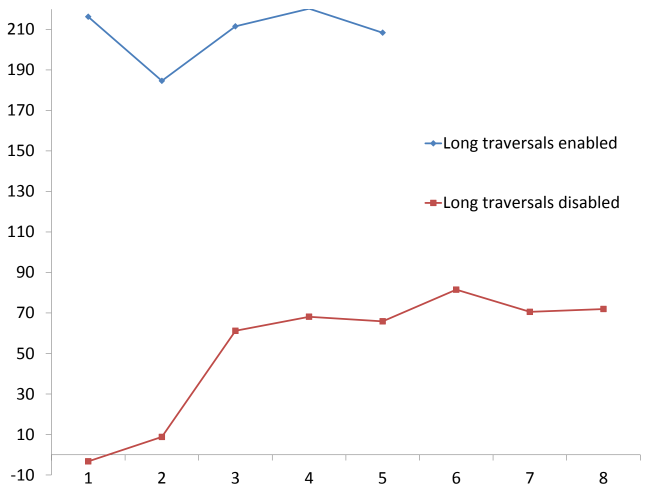

Figure 8. C# STMBench7. The y-axis is percent increase in operations per second, x-axis is number of threads.

this and must needlessly log and validate all the read/write operations involved. This causes massive slow downs. To avoid this we re-factored LeeTM to allocate transient data structures outside of transactions (Opt version). Moreover, LeeTM uses a 3-dimensional array and the Bartok STM does not optimize logging for multidimensional arrays. To avoid the associated overhead, we changed the array to a one-dimensional row-major representation (RM version). LeeTM comes with four workloads: sparseshort, sparselong, memboard and mainboard. We were only able to run sparseshort on the original Bartok STM implementation.

Fig. 9 shows the time elapsed for the different implementations and Fig. 10 shows the relative speedup. The Opt change has a bigger improvement than the RM change. Both does better than either alone. RM benefits the most from our modifications to Bartok, 60-70% with multiple threads, as it has most logging from all the excess logging in the initialization (effectively it is a much larger transaction). Table 5 shows that RM has roughly 35x logging compared to the Opt and OptRM versions, mostly writes. OptRM averages 7% improvement. Opt averages 1% improvement.

5.4 WormBench

We ran WormBench with and without our optimization and observed no significant change in performance. According to [12], the maximum number of reads/writes per transaction is 79/67. A log chunk can typically hold 512 log entries, which is more than sufficient for WormBench transactions. Each Transaction Log Manager maintains a log chunk for reuse over the lifetime of the thread. Therefore no additional log chunks were allocated or released/collected. Fig. 11 shows that we have a speed up of up to 21% for board size of 256, as well as a slow down of the same amount for 512. A board of size 1024 incurs a slow down of 21% at 3 threads and a speed up of 20% at 7 threads. 10 out of the 40 data points are zero (no speed up or slow down). The overall graph shows a total slow down of 4%, due to the extra allocation check.

	Read	Write	Total
Opt	20014	2691	22705
OptRM	16479	2690	19169
RM	17079	722690	739769

Table 5. LeeTM. Logs by type per average transaction.

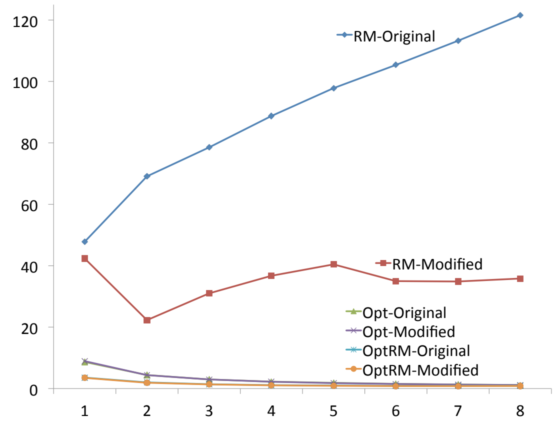

Figure 9. LeeTM. The y-axis is total time and the x-axis is number of threads. Original and modified refer to the STM version. RM, Opt and OptRM refer to, respectively, the row major array optimization, the allocation optimization, and both optimizations.

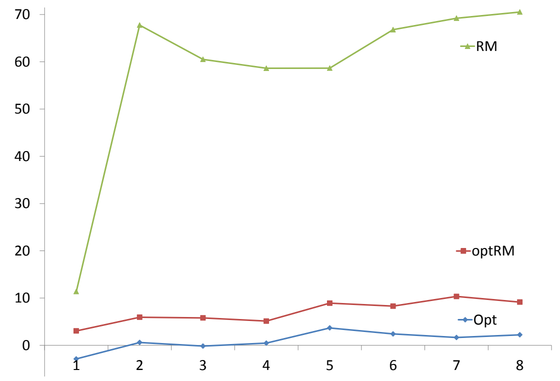

Figure 10. LeeTM. The y-axis gives percent of time saved and the x-axis is number of threads.

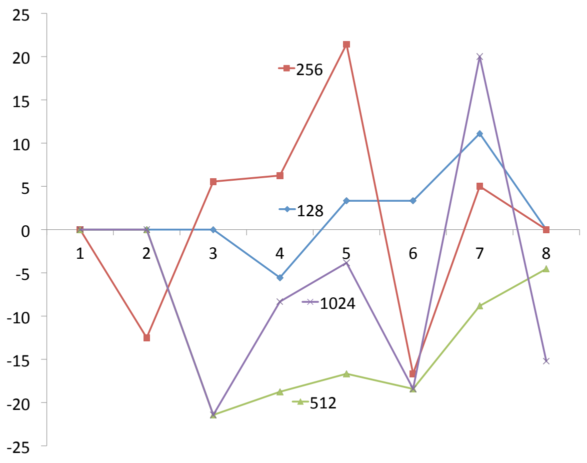

Figure 11. WormBench The y-axis gives operations per second, and the x-axis gives number of threads. Lines represents a different grid sizes. For head size 8x8 and tail size of 8x8.

6. Conclusion

Software transactional memory must maintain a number of auxiliary data structures to detect conflicts between concurrent transactions and record enough information to undo the memory effects of an aborted transaction. STMs for managed languages tend to be library-based and to allocate these data structures from the general heap, adding significant pressure to the memory subsystem and causing more time to be spent on automatic garbage collection. However, since the lifetime of these structures is known, their memory can be more efficiently managed by explicit allocate and free operations. STMs integrated with the host language's runtime can thus avoid collecting transactional garbage. The benefit is greater for longer transactions. For transactions with no memory allocation, we will be able to calculate better estimates of the transaction completion time.

References

[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Conference on Programming language design and implementation (PLDI)*, pages 26–37, Jun 2006.

[2] Cliff Click, Gil Tene, and Michael Wolf. The pauseless GC algorithm. In *International Conference on Virtual Execution Environments (VEE)*, pages 46–56, 2005.

[3] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: a benchmark for software transactional memory. In *EuroSys*, pages 315–324, 2007.

[4] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 14–25, Jun 2006.

[5] Maurice Herlihy, Victor Luchangco, Mark Moir, and William Scherer. Software transactional memory for dynamic-sized data structures. In *Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, Jul 2003.

[6] Maurice Herlihy, Mark Moir, and Victor Luchangco. A flexible framework for implementing software transactional memory. In *Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 253–262, 2006.

[7] Benjamin Hindman and Dan Grossman. Atomicity via source-to-source translation. In *Memory System Performance and Correctness*, pages 82–91, 2006.

[8] Guy Korland, Nir Shavit, and Pascal Felber. Noninvasive concurrency with Java STM. In *MultiProg'10*, 2010.

[9] Virendra J. Marathe, William Scherer, and Michael Scott. Adaptive software transactional memory. In *International Symposium on Distributed Computing*, Sep 2005.

[10] Cyprien Noel. Extensible software transactional memory. In *C* Conference on Computer Science and Software Engineering*, pages 23–34, 2010.

[11] Ian Watson, Chris Kirkham, and Mikel Lujan. A study of a transactional parallel routing algorithm. In *Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2007.

[12] Ferad Zyulkyarov, Adrian Cristal, Sanja Cvijic, Eduard Ayguade, Mateo Valero, Osman Unsal, and Tim Harris. Wormbench: a configurable workload for evaluating transactional memory systems. In *Workshop on MEmory performance: DEaling with Applications, systems and architecture (MEDEA)*, 2008.