Dynamic Performance Tuning of Word-Based Software Transactional Memory

Pascal Felber
University of Neuchâtel, Switzerland
pascal.felber@unine.ch

Christof Fetzer Torvald Riegel
Dresden University of Technology, Germany
{cf2,tr16}@inf.tu-dresden.de

Abstract

The current generation of software transactional memories has the advantage of being simple and efficient. Nevertheless, there are several parameters that affect the performance of a transactional memory, for example the locality of the application and the cache line size of the processor. In this paper, we investigate dynamic tuning mechanisms on a new time-based software transactional memory implementation. We study in extensive measurements the performance of our implementation and exhibit the benefits of dynamic tuning. We compare our results with TL2, which is currently one of the fastest word-based software transactional memories.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Algorithms, Performance

Keywords Transactional Memory, Dynamic Tuning

1. Introduction

Transactional memory (TM) has been proposed as a lightweight mechanism to synchronize threads. It alleviates many of the problems associated with locking, offering the benefits of transactions without incurring the overhead of a database. It makes memory, which is shared by threads, act in a transactional way like a database. The main goal is to simplify the development of concurrent applications, which are becoming more widespread because of the increasing shift to multicore processors and multiprocessor systems.

The performance of software TM implementations (STMs) depends on several factors. First, design choices such as word-based vs. object-based, lock-based vs. non-blocking, write-through vs. write-back, or encounter-time locking vs. commit-time locking can have significant impact on the transaction throughput in different settings. Second, configuration parameters related to the TM implementation, such as the number of locks used to handle concurrent accesses to shared data or the mapping of locks to memory addresses, must be finely tuned to optimize performance. They typically vary from one architecture to another, depending on factors such as the CPU or the size of cache lines.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'08, February 20-23, 2008.
Copyright © 2008, Salt Lake City, Utah ACM 978-1-59593-960-9/08/0002. . . $5.00

Most importantly, the quality of design choices and configuration parameters directly depends on the considered workload: for example, the ratio of update to read-only transactions, the number of addresses read or written by transactions, or the level of contention on shared memory locations will all lead to different optimal configurations. For instance, time-based TMs (see Section 2) are very efficient with read-only transactions but they must validate the read sets of update transactions upon commit; in workloads with large read sets, one can tune the STM implementation so as to reduce the validation overhead. There is no "one-size-fits-all" STM implementation and adaptive mechanisms are necessary to make the most of an STM infrastructure.

In this paper, we study the influence of the workloads on different STM designs and configuration parameters. We focus on word-based, time-based STMs. Unlike object-based designs, word-based TMs allow to directly map transactional accesses to the underlying memory subsystem and they can be easily integrated in various programming languages and supported efficiently within the compiler [5, 16]. We present a lightweight and highly efficient lock-based implementation called TinySTM and evaluate its performance when varying the type of workload. We compare our implementation against TL2 [3], one of the fastest word-based STMs, and show that our design performs better in many situations. Our experimental study gives important insights on the performance of word-based, time-based STM implementations. We also introduce novel mechanisms to speed up the validation cost for large read sets without increasing the abort rate.

Based on the experimental results, we identify important tuning parameters that govern the performance of the STM infrastructure. We propose mechanisms to dynamically tune the STM to the workload by adapting its runtime parameters and searching for the configuration that provides the best transaction throughput. Evaluation demonstrates that, starting from an initial configuration, dynamic tuning allows us to quickly reach a good configuration, and that the performance gain can be very important.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 presents our TinySTM word-based STM implementation, studies its performance alongside TL2 under various types of workloads, and motivates the need for runtime tuning. Section 4 describes our dynamic tuning mechanisms and evaluates their effectiveness. Finally, Section 5 concludes the paper.

2. Related work

Transactional memory (TM) has recently seen a lot of interest. Here, we only discuss the work that is closely related to this paper. Broader discussions of related work can be found in [10, 13].

Word-based TMs (e.g., [16, 3, 7]) access memory at the granularity of machine words or larger chunks of memory and keep TM

metadata external (i.e., at other areas of main memory). Object-based TMs (e.g., [9, 10, 8]) access memory only at object granularity and require the TM to be aware of the object associated with every access. In this sense, word-based TMs are more widely applicable, for example in applications that do not explicitly specify associated objects and run in unmanaged environments.

We refer to a TM that is based on a notion of time or progress as a *time-based transactional memory* (TBTM). A global time base is used to reason about the consistency of data accessed by transactions and about the order in which transactions commit. Typically, TBTMs employ optimistic read operations (i.e., read operations are not visible to other transactions) because invisible reads are less expensive than visible reads. TBTMs then guarantee on the basis of their time base that the snapshot that a transaction takes of the transactional memory at runtime is always consistent. TBTMs thus have an advantage over TMs that only ensure the consistency of the transaction at commit time: transactions that work with inconsistent data could, for instance, enter infinite loops or throw unexpected exceptions. Nevertheless, the per-access costs of TBTMs are small.

The first time-based STM design was SI-STM [11], soon followed by LSA [10] and TL2 [3]. McRT-STM [13] was later extended [16] to use the time-based approach and compiler support to provide software transactions for unmanaged environments. The simplest implementation for a global time base is a shared integer counter. On large systems in which contention on this counter results in a significant bottleneck, external clocks or multiple synchronized physical clocks can be used as scalable time bases [12].

All TBTMs can employ a fast path for validating read sets (i.e., checking whether data read in a transaction did not change after reading it and is thus still valid at the current time). The underlying idea is that if time did not advance in the TM, then no update transaction committed changes.[1] RSTM [15] is not a TBTM but uses this fast path to decrease validation overhead. In all TBTMs and in RSTM, time always covers all object, so the fast path can only be used if there were no updates in the whole TM. This decreases the probability that the fast path can be used. We show in Section 3.2 how to increase the probability that the fast path can be used.

In databases, locking at different levels of granularity has been used for a long time [6, 14]. Although TMs also provide concurrency control, the relative costs for locks are much higher in TMs than in databases. Authors of current STMs are surely aware of the tradeoffs when they select parameters such as locking granularity, but we are not aware of any STM or hardware TM that adapts its parameters and strategies according to the workload or to the hardware it is being executed on.

3. A Lightweight STM Design

TINYSTM is a word-based STM implementation that uses locks to protect shared memory locations. Its name stems from the simplicity and performance of our design. TINYSTM uses a single-version, word-based variant of our LSA algorithm [10] (which is very similar to TL2's algorithm [3]) and uses a time-based design. The notion of time to guarantee consistency in STMs was first proposed in [11] and then simultaneously exploited by LSA and TL2 for object-based and word-based designs, respectively. TINYSTM shares many properties with other word-based STMs: in particular with TL2 (several aspects of our STM library were directly inspired by TL2's reference implementation) but also with Ennals' [4] and Saha *et al.*'s [13, 16] designs. However, TINYSTM follows different design strategies on some key aspects.

[1] Update transactions acquire new timestamps on commit and apply this timestamp to the newly created data versions in the TM.

Unlike TL2's reference implementation—but like Ennals and Saha *et al.*'s algorithms—TINYSTM uses encounter-time locking. Yet, like LSA and TL2, our STM is time-based and guarantees that transactions always read consistent memory states.

The reason why we opted for encounter-time locking is twofold:

- First, our empirical observations appear to indicate that detecting conflicts early often increases the transaction throughput because transactions do not perform useless work. Commit-time locking may help avoid some read-write conflicts, but in general conflicts discovered at commit time cannot be solved without aborting at least one transaction.
- Second, encounter-time locking allows us to efficiently handle reads-after-writes without requiring expensive or complex mechanisms. This feature is especially valuable when write sets have non-negligible size.

In addition, we have implemented two strategies for accesses to memory, each with its unique advantages and limitations: with *write-through* access, transactions directly write to memory and revert their updates in case they need to abort; with *write-back* access, transactions delay their updates to memory until commit time. We shall discuss both strategies shortly.

For the sake of simplicity, TINYSTM has been designed in such a way that a transaction never needs to access another transaction's private memory (besides the shared data structures used for concurrency control). Atomic operations and memory barriers are implemented using Hans Boehm's *atomic_ops* library [1]. These design choices make it straightforward to compile TINYSTM on any 32- and 64-bit architecture supported by *atomic_ops*.

3.1 Basic Algorithm

Locks and Versions As most word-based STM designs, TINYSTM relies upon a shared array of *locks* to manage concurrent accesses to memory (see Figure 1). Each lock covers a portion of the address space. In our implementation, we use a per-stripe mapping where addresses are mapped to locks based on a hash function.

Each lock is the size of an address on the target architecture. Its least significant bit is used to indicate whether the lock is owned. If it is not owned, we store in the remaining bits a version number that corresponds to the commit timestamp of the transaction that last wrote to one of the memory locations covered by the lock.

If the lock is owned, we store in the remaining bits an address to either the owner transaction (when using write-through), or an entry in the write set of the owner transaction (when using write-back). In both cases, addresses point to structures that are word-aligned and their least significant bit is always zero; hence it can be safely used as lock bit.

When using the write-back design, the address stored in the owned lock allows a transaction to quickly locate in its write set the updated memory locations covered by the lock, in case they are accessed again by the same transaction. In contrast, TL2 must check upon access to a memory location whether the current transaction did not yet write to this address, which may be costly when write sets grow large (TL2 uses Bloom filters to avoid unnecessary write set traversals). Read-after-write is not a problem when using the write-through design because the memory always contains the latest value written by the active transaction.

Reads and Writes When writing to a memory location, a transaction first identifies the lock entry that covers the memory address and atomically reads its value. If the lock bit is set, the transaction checks if it is the owner of the lock using the address stored in the remaining bits of the entry. In that case, it simply writes the new value and returns. Otherwise, the transaction can try to wait

for some time[2] or abort immediately. We use the latter option in our implementation.

If the lock bit is not set, the transaction tries to acquire the lock by writing a new value in the entry using an atomic "compare-and-swap" operation. Failure indicates that another transaction has acquired the lock in the meantime and the whole procedure is restarted.

When reading a memory location, a transaction must verify that the lock is not owed nor updated concurrently. To that end, the transaction reads the lock, then the memory location, and finally the lock again (obviously, appropriate memory barriers are used to ensure correct ordering of accesses). If the lock is not owned and its value (i.e., version number) did not change between both reads, then the value read is consistent. There is a subtle problem when using the write-through design. Consider a transaction reading the lock; then, another transactions grabs the lock and writes a new value to memory that is read by the first transaction; the second transaction aborts and restores the initial value of the lock; finally, the first transaction reads the lock again and does not detect a concurrent access, although it has read an inconsistent value. To solve that issue, we additionally store in the lock an *incarnation number* that is incremented each time a transaction aborts and allows us to detect concurrent accesses in such scenarios. We use three bits for the incarnation number in our implementation; in the unlikely event that it overflows, we simply obtain a new version number from the global clock. Note that this problem does not apply to the write-back design.

Once a value has been read, we check if it can be used to construct a consistent snapshot. As with LSA, if the version is more recent than the current validity range of the transaction's snapshot, we try to "extend" the snapshot. This consists in verifying that every address in the read set is still valid and not locked by another transaction. If extension succeeds, we can update the end of the snapshot's validity range up to the value of the clock right before extension. Otherwise, the transaction aborts.[3]

Read-only transactions are particularly efficient because we incrementally construct a snapshot that is valid at all times. No validation is necessary at commit time and, hence, we do not need to maintain a read set.

Write-through vs. Write-back The write-through and write-back designs differ in the way updates are written to memory. With write-through access, updates are written directly to memory and previous values are stored in an *undo log* to be reinstated upon abort. With write-back access, updates are stored in a *write log* and written to memory upon commit. Write-through has lower commit-time overhead and faster read-after-write/write-after-write handling; further, it enables various interesting compiler optimizations, e.g., accesses to a previously locked memory location do not need to go through the STM at all. On the other hand, write-back has lower abort overhead and does not require incarnation numbers to guarantee consistent reads, as discussed above.

Memory Management Using dynamic memory within transactions is not trivial with unmanaged languages. Consider the case of a transaction that inserts an element in a dynamic data structure such as a linked list. If memory is allocated but the transaction fails, it might not be properly reclaimed, which results in memory leaks. Similarly, one cannot free memory in a transaction unless one can guarantee that it will not abort. Dealing explicitly with such situations leads to intricate code.

TinySTM provides memory-management functions that allow transactional code to use dynamic memory. Transactions keep track of memory allocated or freed: allocated memory is automatically disposed of upon abort, and freed memory is not disposed of until commit. Further, a transaction can only free memory after it has acquired all the locks covering it as a free is semantically equivalent to an update.

Clock Management TinySTM uses a shared counter as clock, like LSA and TL2. This approach is both simple and sufficiently efficient on SMP architectures. In case the contention on this global counter becomes a bottleneck in large systems, we can use more scalable time bases such as an external clock or multiple synchronized physical clocks [12].

The maximal value of the clock is 2^{31} on a 32-bit architecture, and 2^{63} on a 64-bit architecture.[4] In 32-bit systems with frequent commits, this value can be quickly reached. Therefore, TinySTM provides a simple clock roll-over mechanism: when a transaction detects that the maximal clock value has been reached,[5] it aborts and waits on a barrier until all active transactions have completed their execution. Then, we reset the clock and all version numbers. While this procedure unnecessarily aborts some non-conflicting transactions and prevents progress for a short period of time, its overhead is negligible as it is executed only rarely.

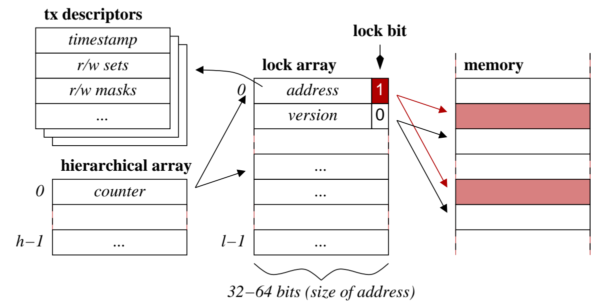


Figure 1. Data structures used in the TinySTM implementation.

3.2 Hierarchical Locking

Transactions that are identified as read-only do not need to keep a read set as the LSA algorithm guarantees that we incrementally construct a consistent snapshot. Update transactions do, however, need to validate their read sets at commit time. This implies that they must verify that all the addresses they have read are still valid, i.e., they are not locked by another transaction and still have the same version number. A notable exception is when the commit time of the transaction is equal to its start time plus one: in that case, validation is not necessary as we know that no other transaction has concurrently written to memory.

Depending on the size of the read set, validation may be costly. A transaction reading a large chunk of memory (e.g., a large matrix) needs to validate every single address read. To speed up validation of large read sets, one can reduce the number of locks, i.e., each lock will cover a larger number of memory locations. However, this can increase the abort rate significantly due to memory operations mistakenly identified as conflicting. To solve this problem, we propose using a hierarchical locking strategy.

[2] Note that the transaction must not wait indefinitely as this might lead to deadlocks.

[3] When keeping multiple versions of every memory location, we could use an old value valid at the time of the snapshot. We do not use this approach in our implementation because, without hardware support, the memory and processing overheads of multi-version designs overcome their benefits.

[4] With the write-through design, maximal values are 2^{28} and 2^{60} as three bits are used for incarnation numbers.

[5] Transactions read the current time when they start. Update transactions additionally obtain the current time when they try to commit.

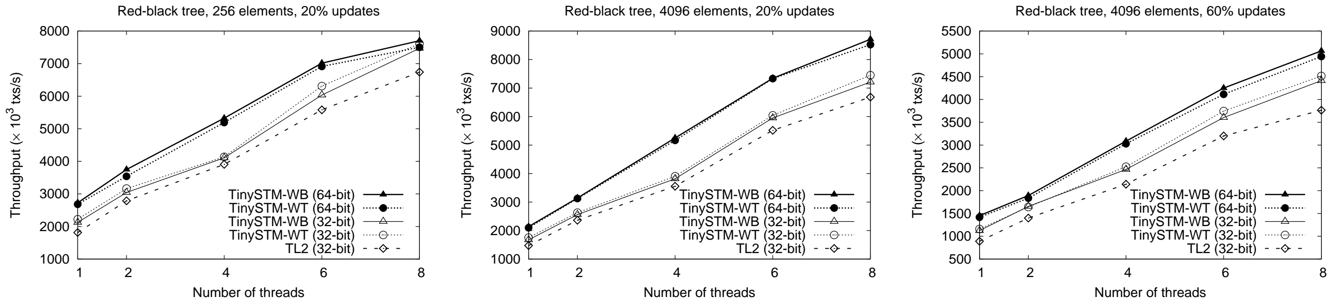


Figure 2. Throughput of the red-black tree.

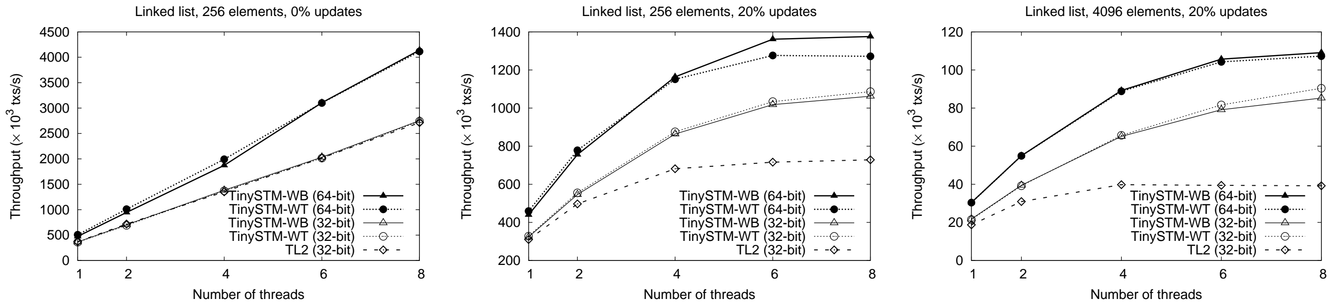


Figure 3. Throughput of the linked list.

In addition to the shared array of ℓ locks, we maintain a smaller *hierarchical array* of $h \ll \ell$ counters (typically 4 to 16) as depicted in Figure 1. Memory addresses are mapped to the counters using a hash function that is consistent with that of the lock array: two memory locations that are mapped to the same lock are also mapped to the same counter. In other words, a counter covers multiple locks and the associated memory addresses. When choosing ℓ as a multiple of h, typically $\ell = 2^i, h = 2^j, i > j$, we can compute the lock index as $(hash(addr) \mod \ell)$ and the counter index as $(hash(addr) \mod h)$.

Each transaction additionally maintains two private data structures: a read mask and a write mask of h bits each. Finally, read sets are partitioned into h independent parts.

When reading or writing a memory location, a transaction will first determine to which shared counter i in the hierarchical array it maps. If the corresponding i^{th} bit in the read mask is zero, then we set it and store locally the current value of the counter. If the memory access is a write, we check the corresponding i^{th} bit in the write mask: if zero, we set it and atomically increment the shared counter. If the memory access is a read and the transaction maintains a read set, we create the new entry in the corresponding i^{th} part of the read set.

Upon validation, we check for every counter i whose corresponding i^{th} bit is set in the read mask if (1) the current value of the counter is equal to the previously stored value, or (2) the current value of the counter is one more than the stored value and the corresponding i^{th} bit in the write mask is set. In either case, we know that no concurrent transaction has locked an address that maps to counter i and we can skip validation of the ith part of the read set (i.e., we can use the validation fast path). By doing so, we are essentially partitioning the locks so that validation can apply to only a portion of the memory locations read by a transaction.

Note that a transaction writing to many locations might need to increment at most h counters. As atomic operations are costly on most architectures, the size of the hierarchical array must be chosen with care: larger h values reduce the validation overhead but may require more atomic operations.

Strictly speaking, the role of the hierarchical array is not to lock memory locations; hence the term hierarchical "locking" is not perfectly accurate. The array does, however, allow transactions to determine whether locks have been acquired. This scheme can be generalized "hierarchically" to multiple levels of nesting.

Obviously, hierarchical locking provides performance benefits only if (1) read set validation is expensive, i.e., update transactions read many memory locations, and (2) there are few writes from competing transactions. However, we believe that these conditions are encountered often enough in real applications for this optimization to be useful.

3.3 Experimental Evaluation

To evaluate the performance of our basic TINYSTM implementation (without hierarchical locking), we have used the same red-black tree benchmark application as used for the evaluation of TL2 in [3]. The code of the red-black tree and TL2 x86 implementations were downloaded from the STAMP distribution [2] and run unmodified.[6]

While the red-black tree is a good example of transactional access to sophisticated data structures, operations on red-black tree typically read and write a small number of memory locations. Therefore, it is not a good indicator of the ability of the STM

[6] The TL2-x86-0.9.0 code is a port to x86 that was *not* performed by the original authors. We had to increase the initial size of TL2 read sets for some benchmarks to work around a bug during expansion. Compilation was performed using the same optimization level for all STM implementations.

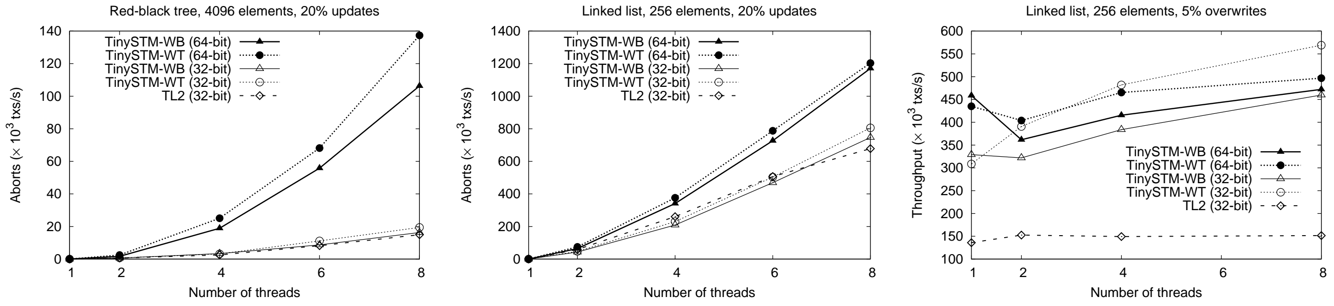


Figure 4. Number of aborts of the red-black tree (left) and linked list (center), and throughput of the linked list with large write sets (right).

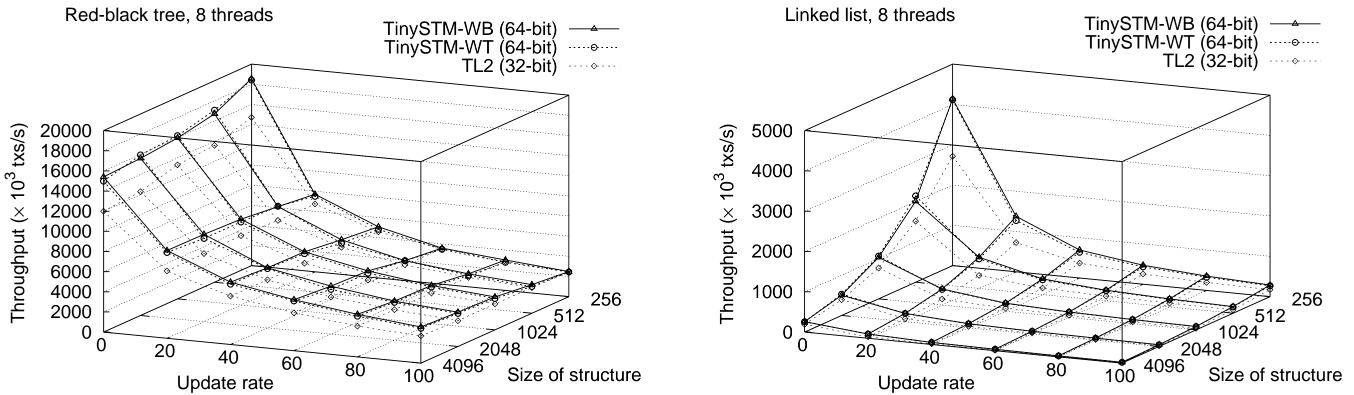


Figure 5. Influence of the size of the data structures and update rates on throughput for the red-black tree and linked list (8 threads).

infrastructure at handling long transactions with sizeable read and write sets. Therefore, we have also experimented with a sorted linked-list implementation used in various other studies (e.g., [9]). The list must be traversed in order to add, remove, or locate entries and read sets can grow large.

The harness application differs from the one in [3] in a number of ways. First, we initially insert a given number of elements in the data structure and we maintain its size almost constant during the experiment. Second, update transactions alternatively add a new element and remove the last inserted element. Therefore, update transactions always write to the data structure (they never fail because of duplicate or missing elements). This allows us to more accurately analyze the performance of the STM as a function of the workload parameters.

All tests were run on an 8-core Intel Xeon machine at 2 GHz running Linux 2.6.18-4 (64-bit). The TL2 x86 port could only be compiled in 32-bit compatibility mode. Therefore, for fairness, we have experienced with version of TINYSTM compiled both in 32-bit and 64-bit mode. 32-bit and 64-bit executables were compiled using gcc 4.0.3 and 4.1.3, respectively. We only compared TINYSTM against TL2, as performance figures of TL2 against other STM designs and hand crafted lock-based code are available in [3].

Figure 2 shows the transaction throughput with the red-black tree and two sizes (256 and $4,096$ initial elements) and update rates (20% and 60%). We observe that all STM designs scale well up to the number of processor cores. The 64-bit versions are more efficient than the 32-bit ones, and TL2 is slightly slower than TINYSTM. The size of the red-black tree does not influence the throughput much. Surprisingly, the throughput is higher with the large tree than the small one for the fastest STMs and 8 threads. The reason is that the small tree produces more contention and, consequently, higher abort rates. Performance drops moderately with higher update rates and scalability is still excellent.

Figure 3 shows the transaction throughput with the linked list and two sizes (256 and $4,096$ initial elements) and update rates (0% and 20%). With read-only transactions, all STMs scale exceptionally well up to the number of cores and all designs perform identically. When increasing the update rate, performance and scalability decrease as conflicts become more frequent. Consider that in the red-black tree, the risk of conflicts is moderate as transactions typically access different subtrees; in the linked list, all transactions access the same set of nodes. The throughput of TL2 suffers from commit-time locking as conflicting transactions often waste time performing lengthy list traversals although they are doomed to abort. With a larger list, performance decreases but the general trends remain the same. We observe in Figure 4 (left and middle) that the number of aborts is indeed much lower with the red-black tree than the linked list.

Update transactions on the red-black tree and linked list have small write-sets and do not allow us to study scalability when writing to many memory locations. To observe such scenarios, we have modified the linked list benchmark so that update transactions

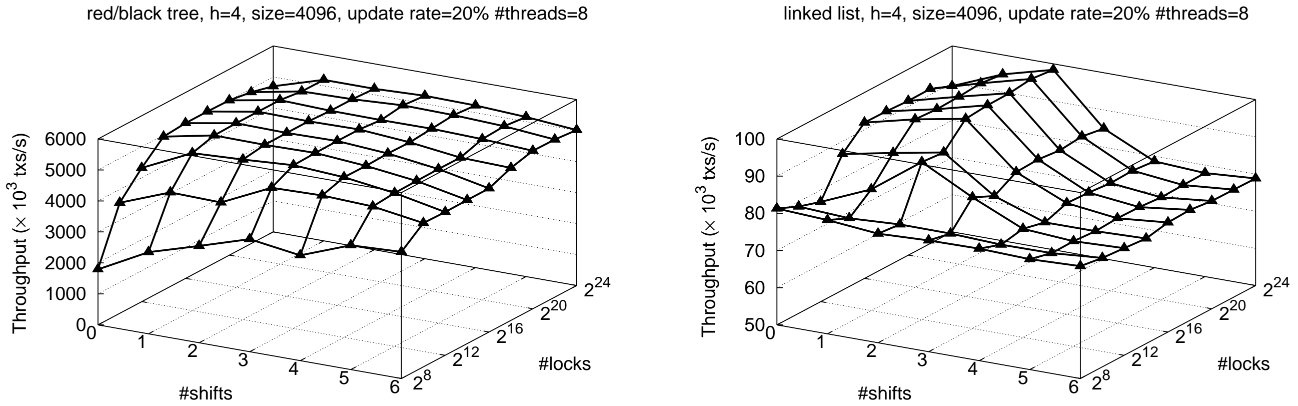


Figure 6. Influence of the number of locks and shifts on the performance of the red-black tree and the linked list.

search for a random value and overwrite any entry encountered while traversing the list up to the random value.

We see in Figure 4 (right) that no STM scales well with this benchmark, even with only 5% update transactions. TL2 appears to suffer from commit-time locking (unsolvable write-write conflicts) and from the overhead imposed by read-after-write and write-after-write detection.

Finally, we have observed the transaction throughput as a function of the size of the data structures and update rates for 8 threads (see Figure 5). Unsurprisingly, one can observe that all STM designs suffer from larger update rates. As could be expected, the influence of the size on the throughput is approximately linear for the linked list and logarithmic for the tree. Finally, one can notice that all STM designs produce the same general shape and that it is difficult to determine which of write-through or write-back is better. In the rest of the paper, we will only use the write-back 64-bit variant.

4. Dynamic Tuning

In TINYSTM, we have various tuning parameters that affect the transaction throughput. The three most important ones are:[7]

1. The hash function to map a memory location to a lock. TINYSTM right-shifts the address and computes the rest modulo the size of the lock array. The number of right shifts (denoted by #shifts) allows controlling how many contiguous addresses will be mapped to the same lock.[8] This parameter allows exploiting the spatial locality of the data structures used by an application.
2. The number of entries in the lock array (denoted by ℓ or #locks). A smaller value will map more addresses to the same lock and, in turn, decrease the size of read sets. It can also increase the abort rate due to false sharing.
3. The size of the array used for the hierarchical locking (denoted by h). A higher value will increase the number of atomic operations but reduce the validation overhead and potential contention on the array's elements. A value of 1 disables hierarchical locking.

To give an indication of the difference in performance between various parameter values, we measured various configurations and workloads using the same experimental setup as in the previous section. Figure 6 shows the change in throughput for the red-black tree and the linked list for an increasing number of shifts and locks. Additionally, Figure 7 shows results for the Vacation benchmark from the STAMP [2] distribution.[9] One can clearly see the influence of these two parameters on performance and observe that it varies significantly depending on the workload.

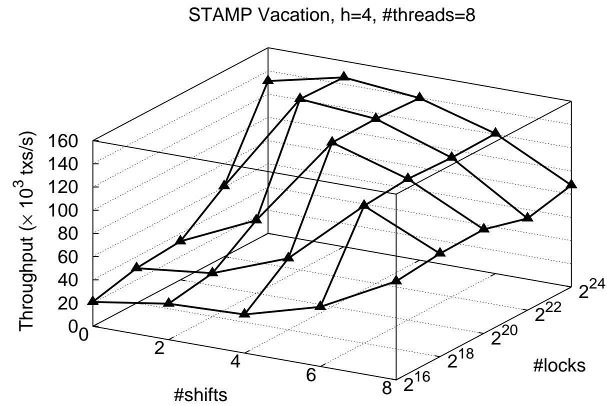


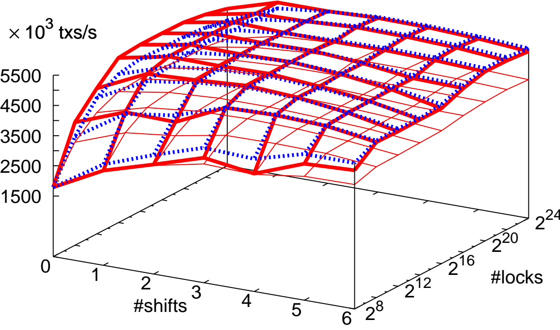
Figure 7. Influence of the number of locks and shifts on the performance of STAMP's Vacation benchmark.

If we also measure the influence of the size h of the hierarchical array, the dependence of the workload becomes even more pronounced: Figure 8 shows that a red-black tree performs best with a small hierarchical array (4 and 16 are better than 64) while the link list performs better with a larger value (i.e., better for 64 than for 4 and 16).

[7] Given its limited impact on performance, as discussed in the previous section, the memory access scheme (write-through vs. write-back) is not considered an important tuning parameter.

[8] Note that these shifts are *in addition* to a right shift of 3 (2 on a 32-bit architecture) to account for word-based memory addressing.

[9] This benchmark was compiled as a 32-bit executable using TANGER [5].



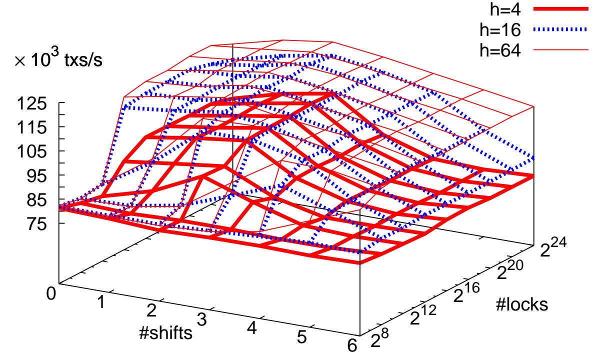


Figure 8. Influence of the size of the hierarchical array on the throughput. The red-black tree benefits from a small hierarchical array while the linked list performs better with larger sizes.

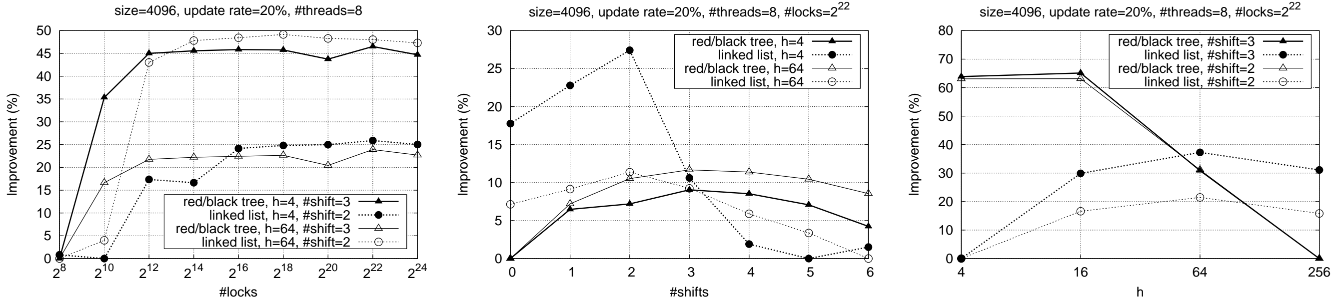


Figure 9. Throughput improvement depending on the number of locks, the number of shifts, and the size of the hierarchical array. The percentage was calculated with respect to the lowest throughput per individual plot.

4.1 Observations

The first observation is that with an increasing number of locks, we get an increase in throughput but eventually the throughput flattens. With a sufficiently high number of locks, it will go down (when consuming too much memory). However, the throughput might have steps (see left graph of Figure 9) and might slightly go down when increasing the number of locks before it goes up again. Also, our measurements indicate that the tuning of the number of locks is quite independent of the other two tuning parameters in the sense that we can tune the number of locks even when using non-optimal tuning parameters for the number of shifts and the size of the hierarchical array. The intuition is that an increasing number of locks reduces false sharing, which is largely independent of the other two tuning parameters. A smaller number of locks could in principle reduce the validation time of an update transaction (because we need to check less locks), but the performance penalty of false sharing dominates.

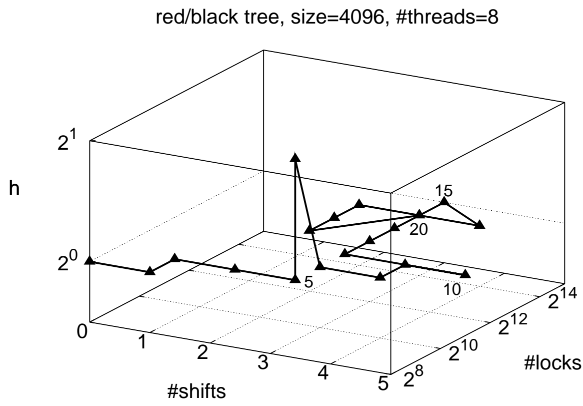
To improve the sharing of locks within a transaction, we have introduced the shift tuning parameter that uses the spatial locality of an application. The number of shifts of the hash function can initially increase the throughput but eventually the throughput will drop again (see middle graph of Figure 9). While there is a possibility that the throughput goes up again after it dropped, we expect that this will not result in an optimal number of shifts. The number of shifts specifies how many consecutive words are assigned to the same lock and, hence, it reflects the spatial locality of the data structures used by the transactions.

The throughput also depends on the size of the hierarchical array (see right graph of Figure 9). Our observation is that the throughput increases initially with the size of the hierarchical array and then drops. The intuition is that a small array limits the overhead of atomic operations on the shared counters and permits a quick check if an update transaction can commit. However, too small an array will result in many false positives, i.e., one has to perform a full validation anyhow, and a large array will suffer from the cost of atomic operations.

4.2 Dynamic Tuning Strategy

While one could manually tune the three tuning parameters for a given workload, it would not be possible to come up with values that will perform well across a large set of workloads. Hence, it is necessary to develop mechanisms to adapt the tuning parameters dynamically.

Our tuning strategy is simple and based on the observations discussed above. We start with a sensible number of locks, 2^{16}; a shift of 0; and a hierarchical array of size 1, i.e., disabled. We then periodically adapt the tuning parameters at runtime. We use the same mechanisms as for clock roll-over to temporarily suspend transactions and update the tuning parameters of TinySTM.



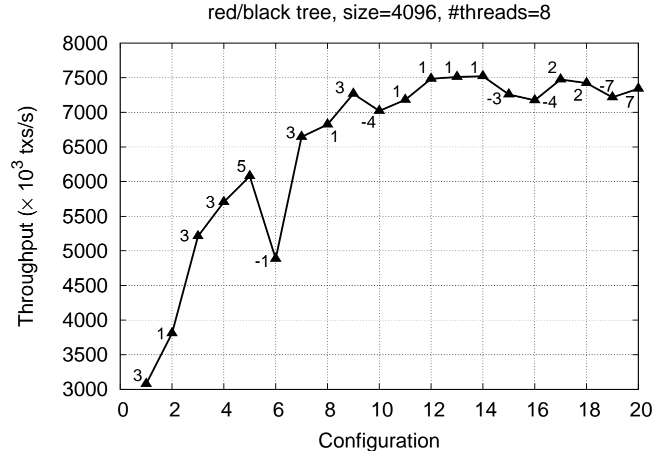
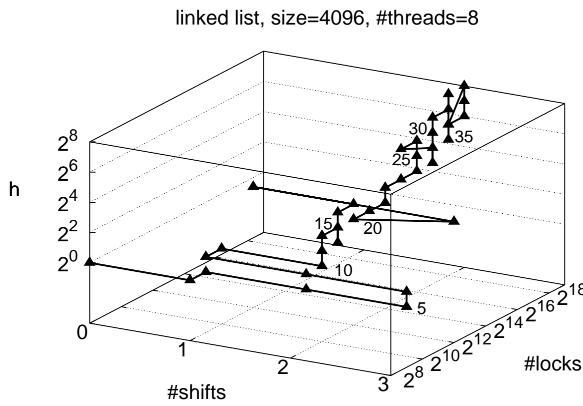


Figure 10. The left graph shows the path taken by the dynamic tuning strategy to optimize the throughput of a red-black tree implementation. The data labels show the configuration numbers. The right graph shows the corresponding change in transaction throughput. The data labels show the moves performed. A move $-x$ denotes a reverse (move 8) followed by move x.



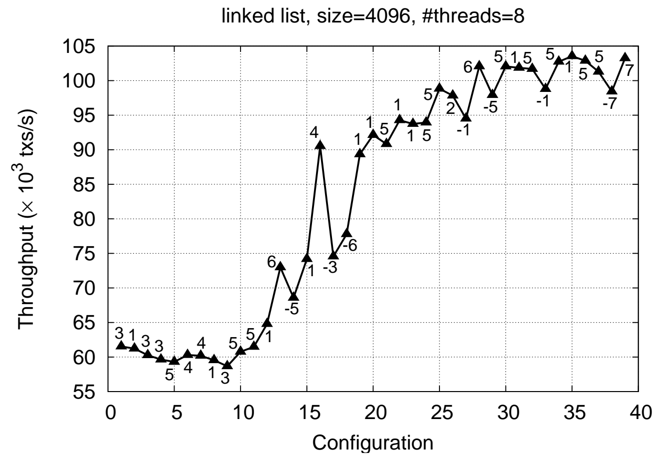


Figure 11. The left graph shows the path taken by the dynamic tuning strategy to optimize the throughput of a linked list implementation. The right graph shows the corresponding change in transaction throughput.

We measure the throughput over a period of approximately one second. Our tuning strategy keeps the most recent throughput for each tuning configuration, where a *tuning configuration* is a triple consisting of the number of locks, the number of shifts, and the size of the hierarchical array. Our tuning strategy has 8 possible moves: (1-2) double/halve the number of locks, (3-4) increase/decrease the number of shifts, (5-6) double/halve the size of the hierarchical array, (7) a *nop* (no change of configuration), and (8) reverse to the configuration with the maximum measured throughput.

Our tuning strategy is a hill climbing algorithm with a memory and forbidden areas. The strategy makes a move and then verifies its effectiveness during the next period. If the performance has decreased by more than 2% or we are more than 10% away from the configuration with the highest throughput so far, it will reverse to that best configuration. If the performance drop is too high, we additionally forbid certain moves: if we increase/decrease the size of the hierarchical array or number of shifts from x to y and the performance drops by more than 10%, we do not further increase/decrease the size or number of shifts beyond x.

We start the tuning process by randomly selecting a move (1-6) and updating the configuration accordingly. After reversing or stopping the hill climbing, our strategy randomly selects another move (1-6) that would lead to a so far uncharted configuration. If all neighbors of the current configuration are explored or forbidden, we reverse to the maximum configuration. If we are already at the maximum configuration, we perform a *nop*. If the throughput drops below that of the second best configuration, we automatically switch to that configuration.

4.3 Evaluation

We have implemented the above tuning algorithm and measured its performance. For testing purposes, we started the measurements with a small number of locks (2^8) instead of a larger number (2^{16}) that we would normally use in TINYSTM. We started with

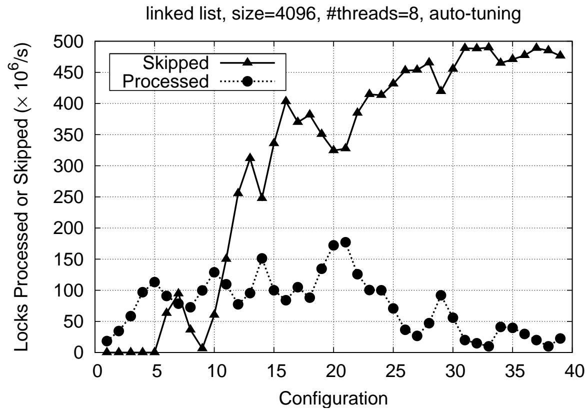


Figure 12. Frequency of locks from read set that need to be processed during validation or can be skipped thanks to hierarchical locking.

a hierarchical array size of 1 and a shift value of 0. The throughput is measured three times in every configuration and the maximum of the three measurements is used as input to the adaptation strategy.

Figure 10 evaluates our red-black tree implementation with auto-tuning: it shows the random path and the associated change in throughput across the configuration space. Interestingly, the tuning algorithm converges to a configuration that has a throughput that is higher than the one we found in our static exploration of the configuration space (see Figure 8); we did not indeed explore the configuration found by the tuning strategy during the static exploration.

Similarly, Figure 11 evaluates our linked list implementation with auto-tuning. The maximum configuration found is close to the one we found in our static exploration. Running the experiments multiple times resulted in different paths but in configurations with similar throughput.

Figure 12 shows that, for the linked list implementation, an increase in the size of the hierarchical array results in reduced validation costs: the number of locks from the read sets that need to be processed during validation drops when increasing the size of the hierarchical array.

5. Conclusion

In this paper we presented the design of TINYSTM, a word-based transactional memory. We showed that TINYSTM provides performance equal to or better than TL2. We also showed that the tuning of TINYSTM for high performance—and we strongly believe that this applies to all STMs—is highly dependent on the given workload. We introduced a simple hill climbing strategy that finds good values for the runtime tuning parameters of TINYSTM. In particular, the auto-tuning found configurations that performed better than the ones we found during a static exploration of the tuning space.

Automatic tuning and adaptivity are especially important given that there is no agreement on what constitutes a typical workload or a good benchmark for transactional memory. They allow us to exploit the full potential of current TM designs, while being ready for workload classes yet to be identified.

TINYSTM is available from `www.tinystm.org` under the GPL v2.

References

[1] Hans Boehm. `http://www.hpl.hp.com/research/linux/atomic_ops/`.

[2] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 69–80. June 2007.

[3] David Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, pages 194–208, September 2006.

[4] Robert Ennals. Software Transactional Memory Should Not Be Obstruction-Free.

[5] Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, and Heiko Sturzrehm. Transactifying Applications using an Open Compiler Framework. In *2nd ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, August 2007.

[6] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. Granularity of locks in a large shared data base. In *Proceedings of the 1st International Conference on Very Large Data Bases (VLDB)*, pages 428–451, September 1975.

[7] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 388–402, October 2003.

[8] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing Memory Transactions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 14–25, June 2006.

[9] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the 22nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, July 2003.

[10] Torvald Riegel, Pascal Felber, and Christof Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, pages 284–298, September 2006.

[11] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot Isolation for Software Transactional Memory. In *1st ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, June 2006.

[12] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based Transactional Memory with Scalable Time Bases. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 221–228, June 2007.

[13] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 187–197, March 2006.

[14] Abraham Silberschatz and Zvi Kedem. Consistency in hierarchical database systems. *Journal of the ACM*, 27(1):72–80, 1980.

[15] Michael F. Spear, Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, pages 179–193, September 2006.

[16] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *Proceedings of the 5th IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, March 2007.